NUCLEAR ENGINEERING

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A SIMULATION MODEL FOR DYNAMIC SYSTEM AVAILABILITY ANALYSIS

by

D. L. Deoss and N. O. Siu

May, 1989

MITNE–287

A SIMULATION MODEL FOR DYNAMIC SYSTEM AVAILABILITY ANALYSIS

by

D. L. Deoss and N. O. Siu

May, 1989

MITNE–287

Nuclear Engineering Department
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

Current methods of system reliability analysis cannot easily evaluate the time dependent availability of large, complex dynamic systems. This report describes a discrete event simulation program developed to treat such problems. The program, called DYMCAM (DYnamic Monte Carlo Availability Model), allows the user to construct system models by specifying components and the links between components. External events, needed in phased mission analysis, are also incorporated. A number of example problems are analyzed to illustrate the accuracy of the base program, and the ease with which various additional features (e.g., complex repair processes) can be incorporated. In particular, an application to a simple process control system is performed to show how continuous variables can be treated within the discrete event simulation framework.

Acknowledgements

This report is based on the master's thesis of the first author. Both authors would like to thank CACI, Inc., who provided the SIMSCRIPT II.5 program language and associated documentation. This language was used to create the DYMCAM program described in the report. Thanks are also given to Professor Tunc Aldemir of Ohio State University for providing data necessary for one of the example problems treated, and to the U.S. Navy, who provided financial support.

TABLE OF CONTENTS

List of Figures

List of Figures (cont.)

List of Tables

<u>List of Appendices</u>

1. INTRODUCTION

Current methods for analyzing the reliability and availability of systems can be characterized as being either static or dynamic. The former include reliability block diagrams [1], fault trees [2], and the GO methodology [3]; these are suited for treating systems whose structures do not change over time. The latter include Markov models (e.g., [4]) and simulation methods (e.g., [5, 6]), and are able to treat time–dependent problems. Methods designed to treat "phased missions" (where the system structure remains constant over a set period of time), such as the GO–FLOW methodology [7], have limited ability to treat changing system structure, and lie somewhere in–between the static and dynamic methods.

Static methods are appropriate for many reliability and availability analysis problems, including the determination of the time–dependent availability of a system consisting of completely independent components. However, if the components interact in a time–dependent manner, dynamic methods are required for an accurate analysis. Such interactions may arise, for example, due to the repair scheme used for components, or due to the behavior of process variables (e.g., when analyzing the reliability of control systems).

The purpose of this report is to present a discrete event simulation model and associated computer code for dynamic system availability analysis. As compared with the more conventionally used Markov modeling approach, this approach has the ability to handle, in a very natural manner, arbitrarily complex problems (e.g., very large numbers of components, non–exponential transition rates, complicated repair strategies). As compared with most other Monte Carlo simulation approaches, the discrete event approach encourages the construction of a model whose elements correspond directly to actual elements in a real system. This leads to a more readily understandable and maintainable model.

The code presented, called the DYnamic Monte Carlo Availability Model (DYMCAM) employs a commercially available simulation language for process–oriented discrete event simulation modeling, SIMSCRIPT II.5 [8]. With the DYMCAM code, the user can construct a system availability model simply by specifying what components are in the system and how they are linked; standard subroutines are used to model component behavior (this is analogous to the decision table approach to fault tree construction [9]). Simple applications of the code are illustrated, as is an extension which allows the treatment of continuous process variables.

Section 2 of this report discusses the discrete event simulation approach, along with the specific characteristics of SIMSCRIPT II.5 used in DYMCAM. It also describes the basic DYMCAM code, including program objectives and assumptions. In Section 3, simple availability problems are analyzed using DYMCAM and results are compared with Markov model results. It is shown that the code predictions are relatively accurate. Section 4 presents a modification to the program to demonstrate the capability of discrete event simulation to model continuous variables. Specifically, the model is altered to perform the storage tank problem described in Ref. 10. Results are compared with a simplified Markov model and the predictions of Ref. 10. Finally, Section 5 summarizes the discrete event simulation approach as applied to dynamic system availability analysis. The advantages discussed include the flexibility and adaptability of the simulation model. The disadvantages include the long running times observed for relatively small numbers of trials. It is pointed out that methods to perform intelligent sampling and to identify key contributors to system unavailability need to be developed to make the approach more practical. These methods may exist for other applications of discrete event simulation; work needs to be done to apply them to availability analysis (which typically deals with rare events).

The source code listing of DYMCAM, as well as sample input and output files, are provided in the report Appendices.

2. DYMCAM DYNAMIC SIMULATION MODEL

Monte Carlo simulation is a potentially attractive method for analyzing the reliability and availability of dynamic systems, due to its ability to treat arbitrarily complex stochastic problems. One possible implementation of the Monte Carlo method in availability analysis is to simulate a discrete time stochastic system in a manner similar to that used for Markov chains. In Figure 1, for example, the probability that the system will transfer from State 1 to State 2 in the next Δt, given that the system is originally in State 1, is approximated by $p_{12} = \lambda_{12}\Delta t$, where λ_{12} may be dependent on a large number of factors (including time). The transition probabilities p_{12} and p_{13} are then used, in a Monte Carlo sampling scheme, to determine (for a given trial) which transition (if any) occurs in the next Δt.

An alternate implementation of the Monte Carlo method is to directly sample the transition times T_{12} and T_{13}. The ordering of the sample results will determine which transition occurs first. This latter implementation focuses on observable quantities (times, rather than hazard rates) and does not require the specification of an arbitrary time scale (the Δt); as a result, it is a somewhat more natural approach and will provide the basis for the DYMCAM (DYnamic Monte Carlo Availability Model) code.

The above description of the second Monte Carlo implementation provides a simple illustration of the discrete event simulation approach used by DYMCAM. More generally in this approach, a queue (sometimes called a "master schedule" or "pending list") is created into which events are entered along with their scheduled occurrence times. For example, a command signal causing a valve to close can be scheduled to occur at a specified time, or a pump could be scheduled to be placed in a standby condition (to simulate the performance of maintenance). At a different time, the valve may be given a command to open or the pump could be placed back in an operational state. Numerous such events can be scheduled and entered in the queue; events in the queue are ordered by their occurrence times.

At the beginning of the simulation, the simulation clock is started and time is advanced to the time corresponding to the first event in the queue. This event is executed (which may result in changes being propagated through the system). Operation continues until there are no more entries in the event queue. The difference between this type of simulation and "continuous simulation" is that in discrete event simulation, it is assumed that no changes occur in the system between the scheduled discrete events.

Note that although Monte Carlo sampling is employed to determine the time intervals in the case of stochastic processes, each sequence of actions is deterministic. Distributions for desired quantities are built up by repeated sampling. It is also important to note that the queue, i.e., the list of actions to be performed, is dynamic; as a result of an action, the queue can be changed. For example, currently scheduled actions can be removed, and new actions added. This list provides a mechanism by which the computer code can treat an arbitrarily complex scenario.

A number of references provide more details on the different approaches to simulation, and on computer languages constructed to implement these approaches (e.g., see [11, 12]). This section discusses the desired characteristics of the dynamic system availability model and the ability of the SIMSCRIPT II.5 language adopted for DYMCAM to provide these characteristics. It also discusses some aspects of the , and the DYMCAM program itself.

2.1 Model Characteristics

Monte Carlo simulation has been used previously in reliability and availability analysis applications. Ref. 13 reviews a number of these applications, including the analysis of fault trees and electric power distribution systems. Ref. 14 outlines an application of discrete event simulation (developed using SIMSCRIPT II.5) to determine the distribution of the time to recover electric power at a nuclear power plant following a loss of offsite power accident. These applications, however, have been developed to solve specific problems. The intent of this work is to take advantage of the characteristics of discrete event simulation to build a more general model which can be applied to a large number of problems.

The characteristics desired of this more general model are:

- Model entities should correspond to physical entities in the system being modeled, where possible.
- Links between entities should also correspond to physical links in the real system.
- Many system models should be constructable simply by selecting component models from an available library of component types (and specifying the links between components).
- Component interactions due to linkages between components should be modeled; interactions due to repair efforts and other operator actions should be easily incorporated.

- Scheduling of system changes at pre–specified times must be possible (e.g., for treating phased missions).
- The model should allow easy updating for incorporating continuous process variables (e.g., for control system analysis).

The first characteristic is desirable more from the standpoint of understandability than efficiency. In it is expected that a model whose basic elements (e.g., subroutines) correspond in a one–to–one manner with the elements of the system being analyzed (e.g., components) will be easier to construct and maintain, perhaps at some cost in execution speed.

The combination of the first three characteristics leads to the specification of "general component models," which consist of specifications of the input to and output from a given component type, and a rule, or set of rules, which determine the component output and state based on input information. Figure 2 shows a general component model. It can be thought of as a box into which signals are fed and from which an output emerges. In addition to signals, information concerning failure and repair rates must be specified. To provide dynamic system information the signals must be able to change value as a function of time.

To allow the propagation of disturbances through a system within the framework of a one–to–one modeling scheme, it is necessary to model links between components. These links consist of the control and process variable signals passed from one component to another. By modeling these signals explicitly, it is possible to create an entire system model out of the general component models. By requiring the components to change state based on their inputs the interaction between components will be modelled. Since in some systems it may be possible to produce loops of elements, it may be useful to continue propagating changes through the system in a cyclic fashion until no further changes occur (otherwise, delays in signal propagation will need to be modeled).

Regarding the fourth characteristic, it is desirable that a model be able to treat groups of related events (i.e., "processes") and their interactions. The last two characteristics in the list indicate the desirability of treating "external events" and continuous variables. Process–oriented modeling allows the integrated treatment of different events in a component's history (e.g., failure and repair), and allows relatively simple treatment of the interactions between components (e.g., one process can interrupt another). External event scheduling allows treatment of events external to the base processes, e.g., the occurrence of a scheduled maintenance outage. Continuous simulation is useful when treating systems whose behavior is strongly affected by the dynamic behavior of process variables (e.g., control systems).

The characteristics described above can be accommodated by a number of languages developed for discrete event simulation. As an example, the process–based simulation modeling approach used in SIMSCRIPT II.5 encourages the definition of "process routines" corresponding to individual components. In the following section, some of the features of the SIMSCRIPT II.5 simulation language are discussed; this provides background needed to better understand the characteristics of the DYMCAM dynamic simulation model.

2.2 SIMSCRIPT II.5

There are many references available describing the SIMSCRIPT II.5 language and related programming techniques for developing simulation models. Ref. 15 is a beginning handbook for understanding the language. For a more detailed description on programming procedures, Ref. 16 should be consulted. Other references used in development of the DYMCAM model include Refs. 8, 17, and 18. All three of these texts provide useful information for understanding the use of SIMSCRIPT commands and modeling techniques.

SIMSCRIPT II.5 is a general programming language which facilitates the development of a discrete–event simulation model. It allows for both process interaction and event–scheduling points of view, or a combination of the two, in simulation modeling. A language extension in current versions allows for continuous simulations [18]. In addition, it also has scientific computing and list processing capabilities. A unique feature of the SIMSCRIPT language is that it can be written in English–like statements.

Several terms are useful to know when attempting to develop an understanding of SIMSCRIPT: scheduling, entity, process, attribute, and sets.

"Scheduling" refers to the discrete event feature of SIMSCRIPT. An event queue is created and events are placed in the queue (scheduled) along with their time of occurrence. The events in the event queue are arranged in the order of their occurrence time and executed in that order. Time then is advanced to the occurrence time of the next event in the queue. The event queue is dynamic; as simulated time progresses, new events may be scheduled and other (previously scheduled) events removed from the queue. For example, a component failure can be scheduled to occur at a certain time. Once the failure has occurred, an event representing repair completion can then be scheduled. As an example of removing events from the queue, an event can be scheduled at the beginning of a simulation which restores all components to as–good–as–new condition at a specified time. This event can remove all scheduled component failures from the event queue. Later in the simulation, the failures can be rescheduled to occur at later times.

An "entity" is a program variable and has a memory location allocated to it once it is created. Entities are of two types, permanent and temporary. Permanent entities are created once, at the beginning of the program, and exist throughout program execution. Temporary entities are created only when needed and memory can be made available again for other variables by destroying the temporary entity once it is no longer needed. This provides a means of keeping data structures contained in computer memory to a minimum, thus providing for more efficient program operation. Several identical entities can be created by using a pointer variable. For example, if a simulation is to contain 10 valves, the following lines of code can be used to create them:

```
reserve pointer(*) as 10
for i equals 1 to 10
do
   create a valve called pointer(i)
loop
```

Then, to refer to valve k, "valve called pointer(k)" can be used in the program.

A "process" is a special SIMSCRIPT entity which has memory associated with it in the same manner as a temporary entity. It can have several identical instances created. For example, if a component is modeled as a process, several identical processes can be created, one associated with each component. The most important feature of a process is that it has a subroutine associated with it which can schedule events and interrupt other processes. A process subroutine can also contain statements which cause the execution of the routine to be suspended, and an event notice to be placed in the event queue to cause the process routine to continue execution at a later scheduled time. If a component is modeled as a process, then the failure of the component can be scheduled by the process and process execution suspended until this time has been reached. Once the failure time has been reached, the component process again begins execution in the line of code following the failure scheduling. Here, for example, a repair delay can be defined and execution suspended until the scheduled delay time has passed. Then repair can be scheduled in the same manner. A process can also create other processes or temporary entities.

All entities and processes can have "attributes" associated with them. This is a way of creating a data structure. For instance, a pump can be defined as an entity. Several pumps may be created. Associated with each pump there may be a demand failure probability, a failure rate, a repair rate, etc. These characteristics can be defined as attributes of the pump entity and thus when a pump is created, memory storage is also allocated for the array of characteristics associated with it. Processes can also have attributes in the same manner.

"Sets" are an important SIMSCRIPT feature. Several items which are of the same type can be grouped as members of a set. These members may be entities or processes, but must be one or the other, in a given set. For example, consider a system containing 100 different input and output signals from ten system components. Several of the signals may be input signals to a given component. A signal set can be defined to group these signals. The set will be "owned" by the component process, and the input signals will "belong" to the set. (In SIMSCRIPT terminology, all sets must have an owner and may have any number of members which belong to the set.)

SIMSCRIPT also has useful statistics features available for evaluating a system simulation. The two basic commands are TALLY and ACCUMULATE. The TALLY command is used to compute statistics of a distribution, such as the mean and variance, at specified instants of time. The distribution can be an array variable. The ACCUMULATE command tracks the behavior of an entity over the duration of a simulation. It performs integration with respect to time and can be used to determine the time–averaged behavior of a system entity. By properly defining the possible system states, this feature can be used directly to calculate the time averaged system unavailability.

The process–interaction approach adopted by SIMSCRIPT is very useful in the analysis of complicated phased mission problems. Components can be modeled as processes, thus allowing each component to control its own time dependent behavior. Failure and repair procedures can be included in the component process subroutine to provide scheduling of failure and repair times. By modeling testing and maintenance as separate processes it is possible to correctly model random testing and maintenance events interrupting component operation and then restarting the components once they are completed.

In addition, if it is desirable to limit repair resources, such as by limiting the number of components under repair at any given time, or if random repair delays are to be incorporated based on the number of components presently failed, the approach can treat this very naturally via a "repair supervisor process." This process could be used to prioritize repair processes by interrupting and rescheduling selected component events. (A purely event oriented simulation approach, which does not group highly related events, would require more effort to implement.)

On the other hand, there are situations where event based simulation is useful (e.g., when dealing with regularly scheduled testing and maintenance). SIMSCRIPT II.5 has the capability to handle these situations; in particular, it has facilities to incorporate "external events," i.e., events whose occurrences are not driven by the simulation model. Finally, SIMSCRIPT II.5 has some capability to perform continuous simulation. This allows analysis of process controls systems, and is demonstrated in Section 4.

2.3 Base Program Characteristics

The DYMCAM (Dynamic Monte Carlo Availability Model) base simulation program was developed with the three primary objectives. These objectives are:

1) the program should enable the user to construct system models for assessing the time–dependent unavailability of dynamic systems,
2) the models should be easy to construct and interpret, and
3) the base program should be easily expandable to incorporate additional features as needed.

The last objective reflects the fact that there are a number of different system characteristics that are more easily treated with modified coding, rather than with user–supplied data.

The following list of characteristics describe some of the key features and limitations of the base DYMCAM program.

1) Failure times are exponentially distributed; repair times are Weibull distributed. Since the SIMSCRIPT II.5 language allows for many types of sampling distributions, it is an easy matter to change distribution types if others are more appropriate for certain applications. These changes can accommodate such time–dependent effects as component aging.
2) Demand failures of active components, valves, and switches are allowed. Data for these failures are entered in the input file and applied to cases of the indicated component failing to transfer in either direction. For instance, a valve can fail to open when it receives a signal to open or it can fail to close once it receives a signal to close. This can be easily generalized via minor changes to the program and the input file.
3) There is no capability to consider delays prior to the start of repair in the base case program listed in Appendix B. However, this can be easily treated by modifying the REPAIR.SUPERVISOR routine, or the process routine associated with a component. If the repair delay acts functionally in the same manner as the delay associated with repair itself, then a simple change in the repair time sampling distribution will suffice.

4) Dependent failure events are considered only to the extent that the loss of the process variable to an active component causes it to fail if it is in an operating state, and external events can be used to model shocks which fail several components simultaneously.

5) Dependent repair events are treated in a problem–specific manner via the REPAIR.SUPERVISOR process subroutine.

6) Uncertainty analysis is not performed.

7) Continuous variables are not treated in the base program. A problem–specific modification designed to demonstrate how continuous variables can be incorporated is described in Section 4. Complex interactions are also considered, to a certain extent in Section 4, as operational states of components are dependent on the level of the continuous process variable.

8) Program output consists of a printout of the time dependent system unavailability (at user–specified time points) and the average system unavailability over the duration of simulated time.

9) Five component types are available to model components. Other component types can be easily created using these five as templates. The component types currently included are: valves, check valves, switches, and generic active and passive components. Component types are defined by the number and type of input signals, by the possible internal states of the component, and by the rules used to process the input/output signals as a function of the component state. A large number of engineering components can be modeled effectively using these basic elements. Active components, valves, and switches have a minimum of three inputs which include a power signal, a command signal, and at least one process input. Passive components have a minimum of one input. They require at least one process input and do not require power or commands. All components can have any number of process outputs. Figures 3–7 provide diagrams and rule tables describing the five component types. The rule tables are taken directly from the program listing of Appendix B. Generally, at the start of a run, no component is initially in a failed state. Note that it is a simple matter to use an external event to change a component to a failed state at time zero.

10) Changes can be forced on the system at any time through the use of external events. These external events can be scheduled to occur during the simulated system operating period and can be used to change the state of components or to change system signals, such as changing a command signal to tell a pump to turn on or off.

The current model requires the times of such occurrences to be known before the start of the simulation and included in the input file. The programming language, however, will allow for the random scheduling of these external events. If this is desirable at a later date, it simply involves creating a process routine (similar to the REPAIR.SUPERVISOR routine) which schedules events in a random fashion.

11) Concerning process signals in the program which represent such system characteristics as fluid flow, pressure, temperature, or electric current, there is no provision in the base model to treat signal magnitudes. It is assumed that the existence or non-existence of the signal is enough to establish the state of components or of the system. In the base program, all components can have any number of process inputs and process outputs. Where inputs are concerned, if the component has at least input signal, then, if the state of the component is correct, all output process signals will be "on". Of course, it is possible to modify the program by changing the input requirements to a component so that it does not produce output unless it has the necessary number of input signals (this is done in a 2-out-of-3 system example in Section 3). This, however, is not a satisfactory solution, in general, if process signal strength is important in the system analysis. More generally, changes can be made to all component routines and the input file to accommodate the notion of signal strength, or "gate" components could be added (this, however, leads to the introduction of non-physical entities in the system model).

2.4 DYMCAM Program Elements and Flow

This section describes the different subroutines in the base version of DYMCAM, and the program flow. The program listing is provided in Appendix B.

In SIMSCRIPT II.5 there are many language features which may not be familiar to those who are accustomed to other programming languages. First of all, every program is composed of many subroutines. Two subroutines which are common to all programs are the "PREAMBLE" and the "MAIN" subroutines.

The PREAMBLE is used to define all program variables and entities used in the rest of the program. The MAIN routine controls overall program execution. It is used to call the subroutines and to start and stop the simulation program. For simple programs, this may be the only routine used other than the PREAMBLE.

The DYMCAM program contains many additional subroutines. Table 1 gives a list of all these routines and their basic purposes. Figure 8 is a flow chart for the program.

Several subroutines are executed before the beginning of actual system simulation. The first of these is the INPUT subroutine. The INPUT routine is used to read the input file and store the information in the appropriate memory locations. In particular, it defines the characteristics of the components to be modeled. This routine is called once during the execution of the program from the MAIN routine.

The next routine called from MAIN is RUN.INITIALIZE. This routine uses the input information to link the system components together. This is done by filing signals in appropriate input and output sets of various components. It also records appropriate signals and components in files associated with each external event for reference when the external event is executed. This routine also initializes all entities. Variables which are not assigned values are automatically set equal to zero by SIMSCRIPT.

The routine TRIAL.INITIALIZE is called from the MAIN program inside the loop which is executed once for each Monte Carlo trial. Its purpose is to reset the state of all components and signals to the initial value they should have at the beginning of execution of the simulation trial.

The next two routines called from inside the loop of the MAIN routine are the scheduling modules. The SCHEDULE.AVAIL.SAMPLES process is used to schedule interrupts in the execution of a simulation run to sample the time dependent system unavailability. The sample times specified by the user are entered in the event queue; the simulation will be interrupted when these times are reached. The actual computing of the availability is done by the AVAILABILITY process. There is a separate AVAILABILITY process created by the program for each time point specified by the input file.

The SCHEDULE.EXTERNAL.EVENTS process is used to schedule the interrupts in the execution of the simulation run for the processing of external events. It schedules these interrupts to occur at the specified times indicated by the input file. For every external event there is an EXTERNAL.EVENT process. Each EXTERNAL.EVENT process has a component set and a signal set associated with it which specify which components and signals are to be changed. The specified changes are performed when the external event is executed and then control is passed to the SYSTEM.UPDATE routine. EXTERNAL.EVENT processes are created by the RUN.INITIALIZE routine along with their associated component and signal files.

Also inside the loop in MAIN is the STOP.SCENARIO routine. It is used to stop the execution of all processes which have not concluded at the end of a trial and to reset the execution of each component to its original operating condition.

The CALL.UPDATE process exists inside the loop of the MAIN routine to escape a complication associated with the program. In SIMSCRIPT, any series of commands executed sequentially without undergoing the simulated passage of time must not contain commands which start and stop the same process or create and destroy the same entity. It is also not possible to activate the same process twice. DYMCAM is designed so that on the initial trial of a run, all component processes are activated at time zero by the RUN.INITIALIZE routine. Thus a notice is put in the scheduled events list which will be executed once the timing routine is begun. One of the first statements in the COMPONENT process is a command to suspend operation, since some components, e.g. standby components, may not be operating at the start of the simulation. Standby components are not allowed to undergo failure in this model and therefore should not have failure times placed in the event queue until they are placed in an operational mode. The components that should be operating are then restarted by the SYSTEM.UPDATE routine.

The problem is that the SYSTEM.UPDATE routine should be executed from the loop of the MAIN routine before the passage of simulated time is begun. This would cause an error since the sequential execution of commands would make it appear that a COMPONENT process has been scheduled to start twice. Therefore the CALL.UPDATE routine is included in the MAIN program loop. Its sole purpose is to wait a short period of time so that the simulation clock is started and all components are in the suspended state before the SYSTEM.UPDATE routine is executed and the operation of selected components is started again.

The SYSTEM.UPDATE routine is called many times during the execution of a simulation program run and it performs many functions. The first time it is called, it is used only to activate the components which should be operational at the beginning of a simulation. These components will advance from their original suspended states and begin their failure and repair cycles. Thus at the beginning of the simulation each operating component, if it has a non–zero failure rate, it will have a failure time scheduled for it in the event queue.

At this point the simulation is started. Currently there are three types of events scheduled in the event queue. These are component failures, availability samples, and external events. The simulation clock will be advanced to the time corresponding to the first event in the queue, the notice scheduling the event will be removed from the overall schedule, and the event will be processed.

If the event is an external event, then an EXTERNAL.EVENT process will be executed. Components in the external event component set and signals in the external event signal set for this external event will be changed to their new values. Then the SYSTEM.UPDATE routine will be called.

If the event is an AVAILABILITY sample, then the system indicator variable, X(t), which indicates whether or not the system is in a satisfactory state, will be tested. The result will be summed with previous and future results for that particular time point, and stored for use in generating the output file. No change to the system is made by this interruption, therefore time is advanced to the next event in the event queue without any changes to the system being performed.

If the event is a component failure, then the COMPONENT process for that particular component will again begin operation. The function FAILURE.TRANSLATION will be called and used to determine the state of the failed component. The failed state will be dependent on the type of component and the initial state, e.g. an open valve will fail closed and a closed switch will fail open. FAILURE.TRANSLATION is an example of the use of the SIMSCRIPT function command which simplifies programming when a series of commands is reused often. The commands in the FAILURE.TRANSLATION function could be placed in the COMPONENT routine without complicating execution of the program. Once the type of failure is determined, a REPAIR.SUPERVISOR process will be activated and the SYSTEM.UPDATE routine will be called.

At this point, the SYSTEM.UPDATE routine is used to propagate changes through the system. It is called any time a component changes state or an external event is activated. It looks for changed signals or components and if it finds a change, it calls the response function (SWITCH, VALVE, etc.) for that particular component or the component which contains the altered signal in its input signal file. If this component changes state, or its output signal changes strength, then it will be necessary to propagate this change through the system. The routine continues to call affected components until no further changes occur. This routine also monitors the overall system state and changes it as necessary to reflect whether the system is available or unavailable as a unit according to the definition provided in the input file.

The SYSTEM.UPDATE routine handles the loops which must occur in a process interaction system. The routine stores the value of all system signals and then looks for changes to this set. If a signal changes value then this is an indication that changes are still occurring in the system. The routine looks for components which have changed state or whose input signals have changed strength and calls the associated response function to ensure the component is in the proper operational state. If it is not, it may change according to its response function and new output signal strengths may be generated. These outputs are inputs to other components, so these components must also be updated. Since the possibility exists for loops to occur in system component structure, once all components have been checked once, the new signals are compared with the old signal strengths. If a difference is indicated, then it is possible that a component is not in its desired state, thus the affected components are evaluated again. This process continues until the value of all signal strengths at the end of an iteration, equal the value of the signal strengths at the beginning of the iteration, indicating that no component has changed state during the last iteration. Since infinite loops may be possible, a maximum number of iterations is specified, which, if exceeded, causes an error message to be printed.

Another important function of the SYSTEM.UPDATE routine is to reset the "failure clock" for components which change state. For example, whenever an ACTIVE component is placed in standby from an operating condition, the COMPONENT process associated with the ACTIVE component is reset so that when it begins operation again it will start a new failure clock. This program feature is very important for the analysis of phased mission problems where it is feasible that a single component may be turned on and off several times during a simulation run.

The five routines entitled ACTIVE, PASSIVE, CHECK VALVE, VALVE, and SWITCH are the response functions called by the SYSTEM.UPDATE routine used to determine the state of all system components and the value of their output signals. These routines are used to change the state of components when a new command is received or the strength of an input signal changes. Each routine tests the state of the component and the value of all input signals and compares the results to a set of control "rules" to determine the new component state and the value of all of the component output signals. If the component is ACTIVE, a VALVE, or a SWITCH and it has been called upon to change state, then the DEMAND.TEST routine is called to determine if the component has failed or not. The DEMAND.TEST routine's sole function is determine if a demand failure occurs based on the demand failure probability for the component. Once the tests are performed and the component state is modified, execution is returned to the SYSTEM.UPDATE routine.

After a component has undergone failure and the effect propagated through the system, the REPAIR.SUPERVISOR routine is called. In the base DYMCAM program, this process is currently used to start a repair process once a component is failed. Thus it simply reactivates the component process which controls the repair time calculation for the component. The repair process is activated from the COMPONENT routine whenever a component fails. The listing of the REPAIR.SUPERVISOR process in Appendix B contains a version which immediately starts a repair once a failure has occurred. Line 31, which causes a Weibull distributed repair delay, is not being used (it is "commented" out). It is used in one of the examples of Section 3. By changing the values of "a" and "b" in lines 23 and 24 it is possible to change the repair delay distribution. However, if different repair delay distributions are desired for different components, then the input file structure and other program characteristics must be changed slightly.

The REPAIR.SUPERVISOR process can also be modified to limit the amount of repair resources available. It is a simple matter to count the number of components failed and the number of components under repair by checking the status variable associated with each component. Then, if too many components are failed, repair of some components could be delayed until repair is finished on other components. It is possible to prioritize repair based on which component has been failed the longest since when a component fails its failure time is recorded. This or any other prioritization scheme can be programmed in to the REPAIR.SUPERVISOR process.

The COMPONENT process is used to control the transfer between good and failed states for all components of the system. There is a COMPONENT process for each system component and these COMPONENTs are created by the RUN.INITIALIZE routine. Within the COMPONENT process there is a section which controls the transfer from operational to failed and a separate section which controls the transfer from failed to operational. Whenever a component changes state the SYSTEM.UPDATE routine is automatically called to propagate the component change through the system as discussed above. Under the current program structure, when a component changes state from operational to failed, the component goes to a suspended state. The repair process is not begun until the REPAIR.SUPERVISOR process reactivates the component.

Once the STOP.SCENARIO event is reached in the event queue, the STOP.SCENARIO process is executed. This process removes all remaining events from the event queue and resets all component processes so that all system processes are ready to begin operation for the next trial. With no events now remaining in the event queue, operation of the program is returned to the MAIN routine which causes the RUN.OUTPUT routine to be called. The RUN.OUTPUT routine is used to write the program results to an output file. The results provided are of two types. There is a print out of the time dependent unavailability data and there is a list of the average system unavailability distribution. Examples of output files are included in Appendix E and are discussed in Sections 3 and 4.

Table 1. DYMCAM SUBROUTINES

Subroutine	Description
PREAMBLE	Defines all Entities and Processes
MAIN	Controls overall execution
ACTIVE	Controls active components
AVAILABILITY	Process that takes time–dependent data for unavailability
CALL.UPDATE	Process that causes delay then calls Update routine
CHECK.VALVE	Controls Check Valves
COMPONENT	Process to control failure and repair of Components
DEMAND.TEST	Determines failure on demand
EXTERNAL.EVENT	Process to execute External Events
FAILURE.TRANSLATION	Function to determine failed state
INPUT	Reads input file
PASSIVE	Controls Passive components
REPAIR.SUPERVISOR	Process to allocate Repair resources
RUN.INITIALIZE	Initializes Variables for Run
RUN.OUTPUT	Prints output results to a file
SCHEDULE.AVAIL.SAMPLES	Process to cause recording of time dependent unavailability data
SCHEDULE.EXTERNAL.EVENTS	Process to schedule External Events
STOP.SCENARIO	Stops execution of all processes
SWITCH	Controls Switches
SYSTEM.UPDATE	Propagates Component changes through the system
TRIAL.INITIALIZE	Initializes Variables for a Trial
VALVE	Controls Valves

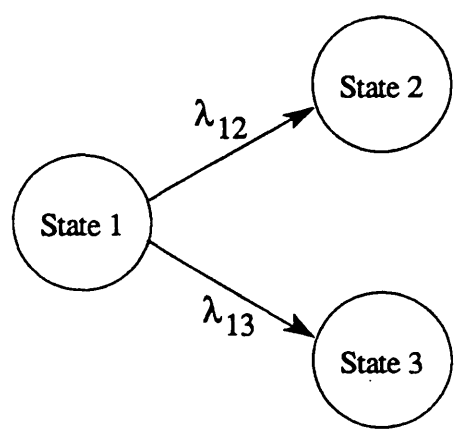


Figure 1. STATE TRANSITION DIAGRAM FOR A SIMPLE SYSTEM

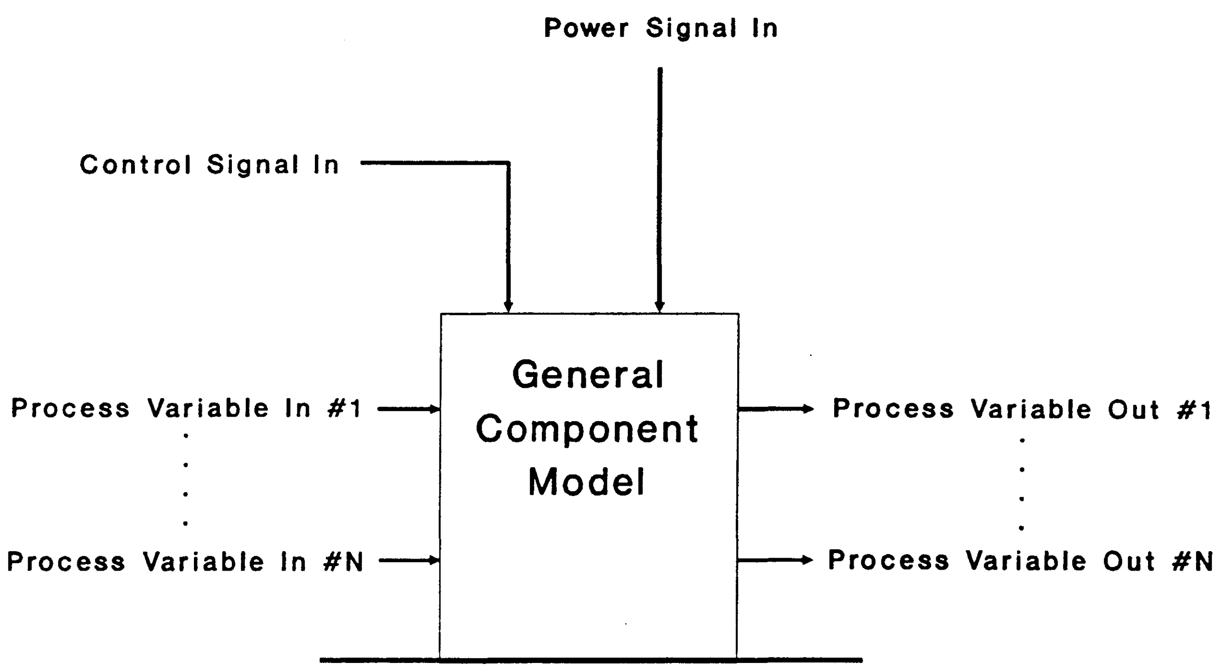
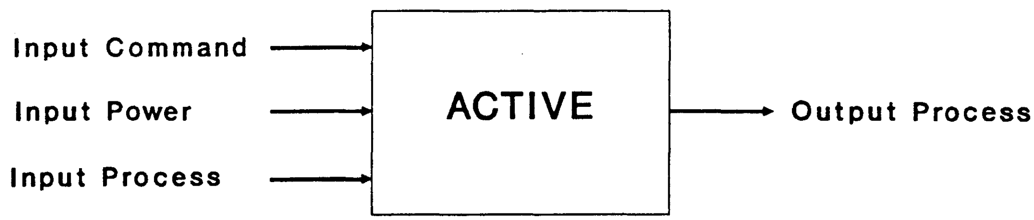


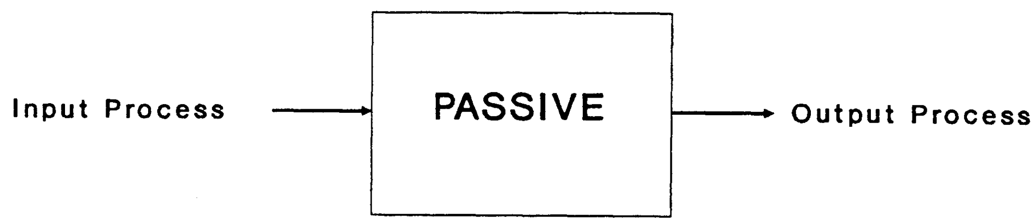
Figure 2. GENERAL COMPONENT MODEL



Decision Table

Case	Command Input	Power Input	Process Input	Initial State	Final State	Process Output
1	-	-	-	failed	failed	no
2	-	no	-	standby	standby	no
3	stop	yes	-	standby	standby	no
4	none	yes	-	standby	standby	no
5	start	yes	no	standby	standby*	no
					failed	no
6	start	yes	yes	standby	standby*	no
					operating	yes
7	-	no	-	operating	standby	no
8	stop	yes	no	operating	failed	no
					standby	no
9	stop	yes	yes	operating	operating*	yes
					standby	no
10	none	yes	no	operating	failed	no
11	none	yes	yes	operating	operating	yes
12	start	yes	no	operating	failed	no
13	start	yes	yes	operating	operating	yes
14	-	-	-	standby*	standby*	no
15	-	no	-	operating*	operating*	no
16	-	yes	no	operating*	failed	no
17	-	yes	yes	operating*	operating*	yes

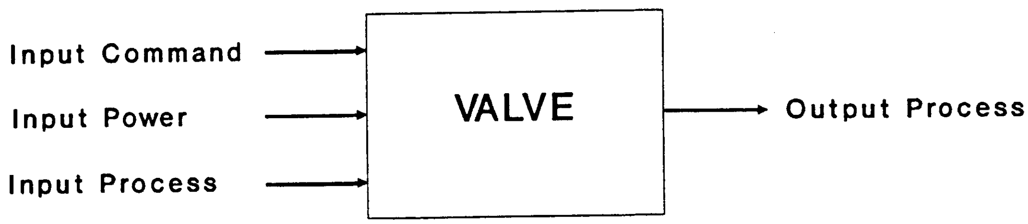
Figure 3. ACTIVE COMPONENT



Decision Table

Case	Process Input	Initial State	Final State	Process Output
1	-	failed	failed	no
2	no	standby	standby	no
3	yes	standby	failed	no
			operating	yes
4	no	operating	standby	no
5	yes	operating	operating	yes

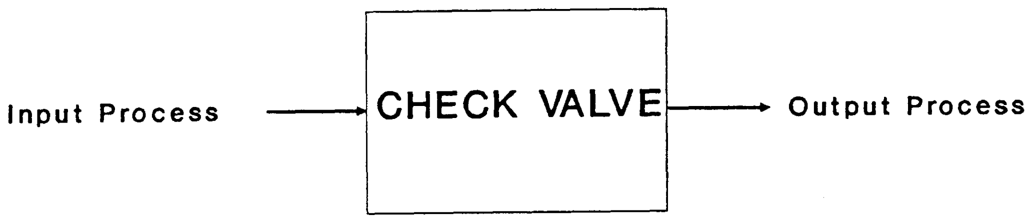
Figure 4. PASSIVE COMPONENT



Decision Table

Case	Command Input	Power Input	Process Input	Initial State	Final State	Process Output
1	-	-	-	failed_open	failed_open	no
2	-	no	-	open	open	no
3	open	-	-	open	open	no
4	none	-	-	open	open	no
5	close	yes	no	open	failed_open	no
					closed	no
6	close	yes	yes	open	failed_open	no
					closed	yes
7	-	-	no	failed_closed	failed_closed	no
8	-	-	yes	failed_closed	failed_closed	yes
9	-	no	no	closed	closed	no
10	-	no	yes	closed	closed	yes
11	open	yes	no	closed	failed_closed	no
					open	no
12	open	yes	yes	closed	failed_closed	yes
					open	no
13	none	-	no	closed	closed	no
14	none	-	yes	closed	closed	yes
15	close	-	no	closed	closed	no
16	close	-	yes	closed	closed	yes

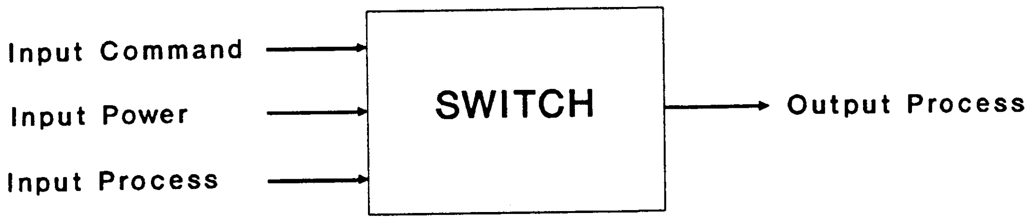
Figure 5. VALVE



Decision Table

Case	Process Input	Initial State	Final State	Process Output
1	-	failed_closed	failed_closed	no
2	no	closed	closed	no
3	yes	closed	failed_closed	no
			open	yes
4	no	failed_open	failed_open	no
5	yes	failed_open	failed_open	yes
6	no	open	failed_open	no
			closed	no
7	yes	open	open	yes

Figure 6. CHECK VALVE



Decision Table

Case	Command Input	Power Input	Process Input	Initial State	Final State	Process Output
1	-	-	-	failed_closed	failed_closed	no
2	-	no	-	closed	closed	no
3	close	-	-	closed	closed	no
4	none	-	-	closed	closed	no
5	open	yes	no	closed	failed_closed	no
					open	no
6	open	yes	yes	closed	failed_closed	no
					open	yes
7	-	-	no	failed_open	failed_open	no
8	-	-	yes	failed_open	failed_open	yes
9	-	no	no	open	open	no
10	-	no	yes	open	open	yes
11	close	yes	no	open	failed_open	no
					closed	no
12	close	yes	yes	open	failed_open	yes
					closed	no
13	none	-	no	open	open	no
14	none	-	yes	open	open	yes
15	open	-	no	open	open	no
16	open	-	yes	open	open	yes

Figure 7. SWITCH

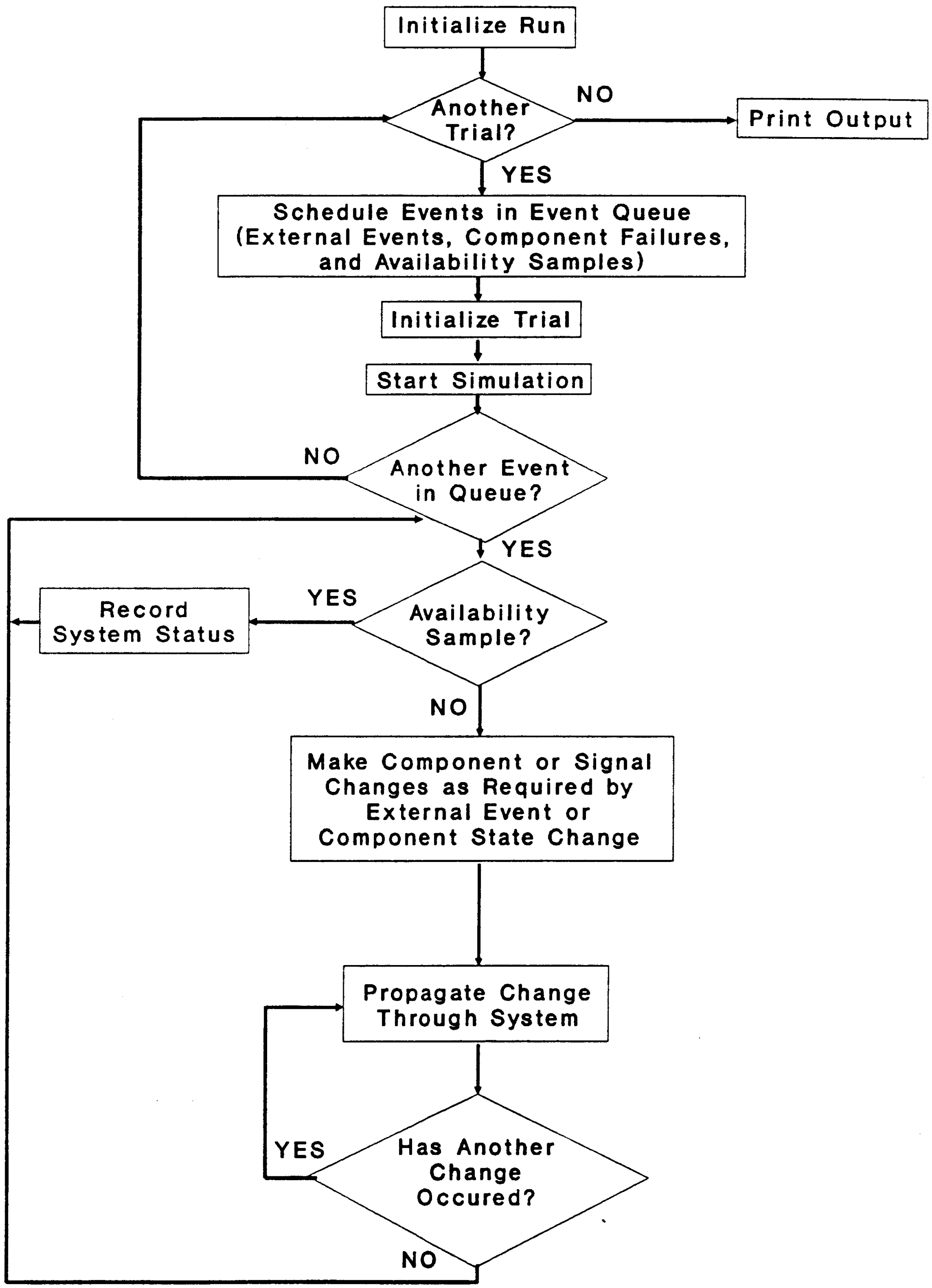


Figure 8. DYMCAM PROGRAM FLOW CHART

3. APPLICATION OF DYMCAM

In this section, a number of simple problems are analyzed to demonstrate the application of DYMCAM. The first problem considered involves a single component with exponential repair and failure times. The second example also involves a single component with exponential repair and failure; in addition, it includes a second repair state which also has an exponential transition time. The third problem involves three pumps in parallel, in series with a valve. Success of the system requires two of the three pumps to operate and the valve to be open. The final example involves a phased mission problem.

The results obtained using DYMCAM are compared with analytical results in the first two examples. A fourth order Runge–Kutta method, obtained from Ref. 19, is used to provide the "exact" answer for the two–out–of–three system, since this problem involves 16 different system states. The phased mission example is compared with exact results as computed using the GO–FLOW method [7].

The chapter concludes with a summary of the performance of the basic DYMCAM dynamic simulation model over the test cases considered. General comments are made concerning the program capabilities, the accuracy of results, and how this approach compares with other system reliability analysis methods.

3.1 Single Component, Single Repair State

The first example problem to be tested using the DYMCAM program is a very simple example involving a single component subject to exponential failure and repair (i.e., the failure times and repair times are exponentially distributed). The time–dependent unavailability of the component is easily obtained using a two–state Markov model:

$$Q(t) = \frac{\lambda}{\lambda + \mu} - \frac{\lambda}{\lambda + \mu}\exp\{-(\lambda + \mu)t\} \tag{1}$$

where λ and μ are the failure and repair rates, respectively. Rather arbitrarily in this example, it is assumed that λ and μ are equal. The asymptotic value of system unavailability is clearly 0.5 since the component will spend equal time in the good and failed states.

The DYMCAM program computes both instantaneous unavailability of a system to provide the dynamic output, and it computes the average unavailability. Instantaneous availability is computed by stopping the simulation (during each Monte Carlo trial) at a user–specified time and checking the system to see if it is in a failed state. A success state is indicated if the system indicator variable is equal to one, and failure is indicated by a zero. The system indicator value is summed over all of the Monte Carlo trials for each selected time point, and divided by the number of trials. The estimate for system unavailability is obtained by subtracting the availability estimate from one.

Average unavailability is calculated over the duration of a simulation. Consider the time line of Figure 9. Since the height of the line in Figure 9 is one, the area under the curve simply equals the total time during the simulation for which the system was unavailable. By dividing this result by the total simulation time, an estimate of the average unavailability is obtained. (Note that the ACCUMULATE function provided by SIMSCRIPT allows easy computation of this result.) For each trial, the unavailability estimate will be slightly different; DYMCAM computes the estimate mean, variance, and selected percentiles of the estimator distribution.

To perform the test for proper asymptotic results, the failure and repair rates were chosen to be 0.01 per hour. Thus after approximately 200 hours the system will have reached its asymptotic condition. Each simulation run covers 10,000 hours. For the simple system only 100 Monte Carlo trials were run to give satisfactory results. To show the fluctuations in unavailability about the asymptotic value, the system instantaneous unavailability was printed at every 500 hours of the simulation. To see the average system unavailability the time averaged system unavailability for each trial was printed.

Table 2 shows the fluctuation of the asymptotic system unavailability estimates about the exact value of 0.5. Over the relatively small number of Monte Carlo trials performed we see that there is a rather large fluctuation. This can readily be reduced by increasing the number of trials since the standard deviation of the estimate decreases as one over the square root of the number of trials.

Figure 10 shows the estimates of the time averaged unavailability for each of the 100 Monte Carlo trials. This figure portrays almost the same information as Table 2. The difference is that Table 2 provides data that was computed using the instantaneous unavailability estimation procedure discussed in conjunction with Eq. (1) and Figure 10 shows the distribution of the time averaged unavailability estimator. The exact average unavailability can be found using (for a specified interval [0,T])

$$A = \frac{1}{T}\int_0^T A(t)dt \tag{2}$$

and where A(t) is given by Eq. (1). Doing this integration, where T = 10,000 and $\lambda = \mu = 0.01$, the result is 0.4975. This result agrees within less than one percent with the mean value of the distribution shown in Figure 10. The standard deviation of the distribution is 0.05. For many applications this deviation is insignificant. Of course, the standard deviation can be reduced by increasing the number of Monte Carlo trials performed.

To check the accuracy of the DYMCAM estimates for time dependent unavailability, another test was run with the same example problem, but over a simulated time period of 200 hours. The number of Monte Carlo trials was increased to 1000. The results are plotted in Figure 11 with the analytic results obtained from Eq. (1).

Figure 11 shows that the simulation model provides good time dependent results for this example. At large values of time, however, it is seen that the simulation starts to deviate from the desired results. For times greater than 200 hours, the simulation continues to fluctuate above and below the exact unavailability. The fluctuations are smaller the larger the number of trials used.

It should be pointed out that a major concern with a simulation approach to systems reliability analysis is the computer time required to perform the analysis. For this simple one component system, the time required to obtain the above results was approximately 30 minutes on an IBM compatible XT machine running at 7.16 MHz. The average unavailability test required a large amount of time due to the long simulated time period of 10,000 hours, which allowed for an average of fifty failure and repair cycles per Monte Carlo trial. (The value of fifty is assumed since if the mean failure and repair times are both equal to 100 hours, then the component will, on the average, go through a complete cycle of failure and repair every 200 hours.) The time dependent analysis required 30 minutes to run even though it simulated a shorter time period, because the unavailability of the system was sampled once every simulated hour (200 points) which slowed down program execution. The program runs in about one sixth the time on a COMPAQ 386SX machine. Methods of reducing computer time required are discussed in Section 5.

3.2 Single Component, Dual Repair State

The second example problem is an extension of the first; here, the component is forced to wait for a random amount of time (exponentially distributed), prior to repair. This example partially demonstrates the capability of the REPAIR.SUPERVISOR routine (a subroutine in the DYMCAM program that determines when component repair is initiated) to treat more complicated repair strategies; a more complete exercise would involve the interaction of multiple components undergoing repair (where one repair process could interrupt the other). This example also demonstrates the ease at which the DYMCAM program can be modified to meet specific applications.

In Appendix B the entire program listing for DYMCAM is shown. In the REPAIR.SUPERVISOR process routine, Line 31 contains the WAIT command used to simulate delays in the third component state. It has been modeled as a Weibull distributed variable, but by proper choice of the parameters, the Weibull distribution becomes an exponential distribution. The Weibull cumulative distribution function is given by:

$$F_T(t) = 1 - \exp\left\{-\left[\frac{t}{\beta}\right]^{\alpha}\right\} \tag{3}$$

where α and β are the distribution parameters. By letting the parameter α equal 1.0, the Weibull distribution becomes an exponential distribution with hazard rate equal to $1/\beta$. Lines 23 and 24 of the REPAIR.SUPERVISOR routine define the exponential distribution with a mean failure rate of one failure every 100 hours. If, in the future, it is desirable to enter different delay distributions for various components, the parameters for the Weibull distribution can be read in the INPUT routine in the same manner as the repair distribution parameters.

The failure and repair rates for this example were chosen to be the same as for the first example. Thus, with a mean repair delay time of 100 hours, the component now has three equal transfer rates from its three states. Thus it is evident that for the asymptotic case, the component will spend equal time in each of the three states. The component is only available when it is in its operational state, thus the asymptotic unavailability is 0.6667.

To test the asymptotic unavailability estimates developed by DYMCAM, the program was run for a simulated component operation of 10,000 hours and 100 Monte Carlo trials. As in Example 1, the component was modeled as a passive element, although results would be the same for modeling the component as any of the other four component types for this simple case. Again the unavailability was sampled at 500 hour intervals to show the fluctuation of the value around the expected value of 0.6667; Table 3 shows the results.

For this test the average system unavailability was also printed out for each of the 100 Monte Carlo trials. The range of values was divided into nine bins and the number of trials in each bin plotted against the central unavailability value for that bin. The results are shown in Figure 12. The exact result for the average unavailability is found to be

0.6634. (This indicates that the first 200 hours of operation do slightly lower the result.) The simulation result agrees with the exact result within less than one percent difference. Again the standard deviation of the simulation result is 0.05 which is insignificant for many analyses.

To compute the time dependent unavailability of this component, the simulation time was reduced to 200 hours, and the number of trials increased to 1000 to reduce the variance of the results. Unavailability samples were taken every simulated hour and the results are plotted in Figure 13. For this example it is also possible to derive the analytic equations for the probability that the system is in any one of its three states using a Markov modeling. The three equations are:

$$\begin{aligned} \frac{dP_0}{dt} &= -\lambda P_0 + \mu_2 P_2 \\ \frac{dP_1}{dt} &= -\mu_1 P_1 + \lambda P_0 \\ \frac{dP_2}{dt} &= -\mu_2 P_2 + \mu_1 P_1 \end{aligned} \tag{4}$$

where P_i represents the time–dependent probability that the system, is in the ith state.

Rather than solve these equations using Laplace transforms or matrix exponentiation techniques, a fourth order Runge–Kutta numerical integration routine taken from Ref. 19 was used. The component unavailability was calculated using $1 - P_0(t)$. This result is plotted in Figure 13 for comparison with the simulation results.

From Figure 13 it is seen that the simulation program again gives good results for the time dependent unavailability. As the value of simulated time increases there is a fluctuation of the simulation results about the desired value, but as explained before this can be reduced by increasing the number of trials. The computer time required for these two experiments was comparable with the first example problem (approximately 30 minutes). The addition of the third component state did not significantly alter the time required to complete the run. The most important contributions to running time appear to be the length of simulation time for each trial and the number of time samples taken during each trial (the sampling process interrupts the simulation).

3.3 Two–Out–Of–Three System

The third test case for DYMCAM considers a more complicated system composed of three pumps connected in parallel. Figure 14 shows a diagram of the system. The output of the pumps is fed to a common header where the flow then enters a valve. Success of the system requires at least two pumps to be operating and there to be flow output from the valve.

As discussed in Section 2, the component types in the base DYMCAM program assume that a satisfactory level of signal input exists as long as a single signal input exists. For this example, therefore, a slight modification to the program is made in Line 129 of the VALVE routine. By changing the test to require two input processes, the valve would not have an output unless at least two of the pumps are providing input to the valve. This problem, therefore, illustrates another simple way by which the base DYMCAM program can be modified to suit the needs of a specific problem. Because of the direct correspondence between program entities and physical entities, the modifications are both small and limited in scope.

In this problem, all pumps are chosen to be identical and the valve is modeled with failure and repair rates identical to those of the three pumps. There are four components which can be in either a failed or operational state which means the system can be in $2^4 = 16$ possible states. (Due to symmetry, these states can be grouped into 8; this is not done in this analysis.) Since all failure and repair rates are equal, in the asymptotic case each system state has equal probability of occurrence. Only four of the states correspond to the system being in an available condition, thus twelve states (or three fourths of the states) contribute to system unavailability. Thus, the asymptotic unavailability should be 0.75.

As in the previous two examples, the program was run for a simulated time period of 10,000 hours and for 100 Monte Carlo trials. Again, the failure and repair distributions were chosen to be exponential with mean values of 100 hours. Table 4 shows the fluctuation of unavailability about the exact value of 0.75. The time–dependent analysis described below indicates that the system reaches its asymptotic state after approximately 200 hours. Thus the actual value for average system unavailability should be slightly less than the asymptotic value of 0.75.

The average value of unavailability over the 10,000 hour simulation was printed for each of the 100 trials and the resulting distribution is plotted in Figure 15. This figure indicates that the mean value of unavailability is 0.7428; the standard deviation of the distribution is 0.03.

To determine the time dependent performance of this system, a second run was done over a simulated time period of 200 hours using 1000 Monte Carlo trials. The unavailability was sampled every hour.

For comparison, the system was modeled as a Markov system. The sixteen possible states for this system are:

0	–	All components are good
1	–	Pump #1 failed
2	–	Pump #2 failed
3	–	Pump #3 failed
4	–	Valve failed
5	–	Pumps #1 and #2 failed
6	–	Pumps #1 and #3 failed
7	–	Pump #1 and Valve failed
8	–	Pumps #2 and #3 failed
9	–	Pump #2 and Valve failed
10	–	Pump #3 and Valve failed
11	–	Pumps #1, #2, and #3 failed
12	–	Pumps #1 and #2 and Valve failed
13	–	Pumps #1 and #3 and Valve failed
14	–	Pumps #2 and #3 and Valve failed
15	–	All Components are failed

Figure 16 shows the Markov state transition diagram for this system. All transition time distributions are exponential with characteristic rates of 0.01 per hour. The Markov equations for the system were solved using a fourth order Runge–Kutta numerical integration routine. This exact solution is plotted in Figure 17 along with the simulation results for comparison.

It is seen from Figure 17 that even for this more complicated system, the DYMCAM simulation program provides good results for the time dependent unavailability. Again the fluctuation of the results about the desired result can be seen at larger time values and it is evident that the accuracy of Monte Carlo analysis is directly related to the number of trials performed.

For this example problem, the computer time required to run the 10,000 hour simulation run for estimation of the asymptotic unavailability value was approximately three hours on an IBM compatible XT running at 7.16 MHz. The second run to determine time dependent unavailability required four and one half hours. The significant increase over the time required for the first two tests is due to the fact that this problem is more complicated (sixteen system states as opposed to two or three) which leads to a far greater number of calculations to be performed during execution of the program. The difference between the two times required for the asymptotic run and the time dependent analysis run reflects the larger number of Monte Carlo trials performed and the larger number of program interruptions (for time–dependent availability sampling).

3.4 Phased Mission Problem

The fourth example problem considered demonstrates the phased mission capability of the DYMCAM program. For comparison, this problem is derived from the GO–FLOW example problem discussed in Ref. 7. The solution derived using the methods of Ref. 7 are used for comparison with the results of the simulation method.

The problem to be solved involves a simple electrical circuit. Figure 18 gives a diagram of the system. It is composed of a battery, having a demand failure probability of 0.1, which will supply power to two parallel circuits. Each circuit has a switch and a light bulb. The switches are identical and have a demand failure probability of 0.3. Neither the battery nor the switches are presumed to experience run time failures. The light bulbs in the system are considered identical and they have a 0.2 probability of failing on demand and a run time failure rate with a mean value of one failure every 1,000 hours.

The actual problem solved in Ref. 7 considered that the switches had a probability of premature closure, however in the DYMCAM model this type of failure would be modeled as a run time failure and would mean that there is an equal probability that the switch could open once it is closed. Since the latter condition was not considered in Ref. 7, the premature failure probability was excluded from the simulation analysis.

The phased mission problem to be solved considers that at time zero the battery is connected to the circuit and has a 0.9 probability of being good. A fraction of a second later one of the switches is closed, then ten hours later, the second switch is closed. The analyst wishes to determine the probability that at least one light is on immediately following closure of the first switch (call this time $t = 0.0$), immediately prior to closing of the second switch (time $t = 9.99$ hours), instantly following closure of the second switch (time $t = 10.0$), and twenty hours after closure of the first switch (time $t = 20.0$). Analysis using the DYMCAM program was done varying the number of Monte Carlo trials from 1,000 to 10,000 to investigate the sensitivity of the results.

To solve this problem using the DYMCAM program, the external event feature was used. This capability allows the input file to contain instructions which will cause a signal to change at an instant of time after the start of the simulation. This function was used to give the battery a process signal input at time $t = 0.0$, to give the first switch a command signal to close at time $t = 0.0$, and to give the second switch a command to close at time $t = 10.0$ hours. This feature allows the DYMCAM program to easily solved phased mission problems.

Tables 5 and 6 summarize the results of the ten tests run using the DYMCAM program. Table 5 shows the results using from 1,000 to 5,000 Monte Carlo trials and Table 6 shows the outcome of tests using 6,000 to 10,000 trials. The tables show the actual probability of at least one light being on at each of the four designated time points as calculated using the GO–FLOW method and the corresponding values calculated with the simulation program. The difference of the simulation value from the actual value is shown and the percent error is calculated as the difference divided by the actual value. For an indication of the variance, the number of trials which would need to have been changed to give the actual results are indicated. For example, for the case where t = 20 hours and N = 1,000 trials, Table 5 indicates that –10 trials would have to be changed. This means that 10 of the 1,000 trials for which a light was not on at t = 20 would need to have had a light test on in order for the simulation results to agree with analytic results.

It can be seen in these two tables that the error decreases as the number of trials is increased and for 10,000 trials the percent difference between the actual availability values and the estimates from the simulation program are less than one percent for all time points. As expected, there is very little difference in the error percentages for two cases separated by only 1,000 trials. For example, there is an average of only a 0.5 percent difference between the values for the 3,000 trial case and the 4,000 trial case. The amount of error should decrease with increasing number of trials in proportion to one over the square root of the number of trials and this is evident by comparing the 1,000 and 10,000 trial cases.

The computer time required for these tests was approximately fifty minutes for every 1,000 trials, thus the 10,000 trial case took about eight and one half hours to run. This time requirement refers to an IBM compatible XT running at 7.16 MHz. The approximate time for the 10,000 trial on a 386 personal computer is estimated to be about 1.5 hours.

3.5 Summary

The examples in this section demonstrate the application of the base DYMCAM model, and simple modifications that can be made to extend to base model to particular problems. Regarding the latter, the second example indicates how the REPAIR.SUPERVISOR routine can be expanded to allow more complicated repair processes; the third example demonstrates an application to m–out–of–n systems. In each example, the simulation predictions agreed with the exact results, both asymptotic and time–dependent, quite well. The dependence of simulation accuracy on the number of trials was also verified.

An important result is that the computing time requirement for a DYMCAM simulation is significant. This issue is further discussed in Section 5 of this report. The following section discusses a modification of DYMCAM developed to treat a control system problem.

Table 2

SINGLE COMPONENT, SINGLE REPAIR STATE, INSTANTANEOUS UNAVAILABILITY

TIME	UNAVAILABILITY
0.0	0.0
500.0	0.52
1000.0	0.38
1500.0	0.51
2000.0	0.60
2500.0	0.48
3000.0	0.48
3500.0	0.46
4000.0	0.51
4500.0	0.47
5000.0	0.45
5500.0	0.56
6000.0	0.41
6500.0	0.54
7000.0	0.48
7500.0	0.45
8000.0	0.50
8500.0	0.51
9000.0	0.57
9500.0	0.55
100000.0	0.49

Table 3

SINGLE COMPONENT, DUAL REPAIR STATE
INSTANTANEOUS UNAVAILABILITY

TIME	UNAVAILABILITY
0.0	0.0
500.0	0.63
1000.0	0.70
1500.0	0.70
2000.0	0.68
2500.0	0.67
3000.0	0.65
3500.0	0.67
4000.0	0.72
4500.0	0.69
5000.0	0.64
5500.0	0.59
6000.0	0.63
6500.0	0.68
7000.0	0.65
7500.0	0.68
8000.0	0.69
8500.0	0.68
9000.0	0.61
9500.0	0.70
10000.0	0.64

Table 4

TWO OUT OF THREE COMPONENT INSTANTANEOUS UNAVAILABILITY

TIME	UNAVAILABILITY
0.0	0.0
500.0	0.72
1000.0	0.80
1500.0	0.76
2000.0	0.75
2500.0	0.78
3000.0	0.78
3500.0	0.73
4000.0	0.74
4500.0	0.66
5000.0	0.76
5500.0	0.68
6000.0	0.66
6500.0	0.77
7000.0	0.78
7500.0	0.71
8000.0	0.73
8500.0	0.67
9000.0	0.77
9500.0	0.71
10000.0	0.81

Table 5. LIGHT BULB PROBLEM RESULTS (1,000 TO 5,000 TRIALS)

QUANTITY	ACTUAL	NUMBER OF TRIALS 1000	2000	3000	4000	5000
TIME 0.0 hours						
Result	0.5040	0.4910	0.5070	0.5057	0.5033	0.5020
Difference from actual value	——	–0.0130	0.0030	0.0017	–0.0007	–0.0020
Equivalent number of trials	——	–13.0	6.0	5.0	–3.0	–10.0
Percent Error	——	–2.6	0.6	0.3	–0.1	–0.4
TIME 9.99 hours						
Result	0.4990	0.4880	0.5025	0.5010	0.4985	0.4968
Difference from						
actual value	——	–0.0110	0.0035	0.0020	–0.0005	–0.0022
Equivalent number	——	–11.0	7.0	6.0	–2.0	–11.0
of trials	——	–2.2	0.7	0.4	–0.1	–0.4
TIME 10.0 hours						
Result	0.7236	0.7060	0.7270	0.7320	0.7275	0.7266
Difference from actual value	——	–0.0176	0.0034	0.0084	0.0039	0.0030
Equivalent number of trials	——	–18.0	7.0	25.0	16.0	15.0
Percent Error	——	–2.4	0.5	1.2	0.5	0.4
TIME 20.0 hours						
Result	0.7191	0.6980	0.7205	0.7257	0.7215	0.7212
Difference from actual value	——	–0.0211	0.0014	0.0066	0.0024	0.0021
Equivalent number of trials	——	–21.0	3.0	20.0	10.0	10.0
Percent Error	——	2.9	0.2	0.9	0.3	0.3

Table 6

LIGHT BULB PROBLEM RESULTS (6,000 TO 10,000 TRIALS)

QUANTITY	ACTUAL	6000	7000	8000	9000	10000
		NUMBER OF TRIALS				
<u>TIME 0.0 hours</u>						
Result	0.5040	0.4995	0.4950	0.4971	0.4998	0.5007
Difference from actual value	——	–0.0045	–0.0090	–0.0069	–0.0042	–0.0033
Equivalent number of trials	——	–27.0	–63.0	–55.0	–38.0	–33.0
Percent Error	——	–0.9	–1.8	–1.4	–0.8	–0.7
<u>TIME 9.99 hours</u>						
Result	0.4990	0.4935	0.4889	0.4913	0.4939	0.4948
Difference from actual value	——	–0.0055	–0.0101	–0.0077	–0.0051	–0.0042
Equivalent number of trials	——	–33.0	–71.0	–62.0	–46.0	–42.0
Percent Error	——	–1.1	–2.0	–1.5	–1.0	–0.8
<u>TIME 10.0 hours</u>						
Result	0.7236	0.7238	0.7204	0.7205	0.7214	0.7243
Difference from actual value	——	0.0002	–0.0032	–0.0031	–0.0022	0.0007
Equivalent number of trials	——	1.0	–22.0	–25.0	–20.0	7.0
Percent Error	——	0.03	–0.4	–0.4	–0.3	0.1
<u>TIME 20.0 hours</u>						
Result	0.7191	0.7185	0.7143	0.7145	0.7154	0.7186
Difference from actual value	——	–0.0006	–0.0048	–0.0046	–0.0037	–0.0005
Equivalent number of trials	——	4.0	–34.0	–37.0	–33.0	–5.0
Percent Error	——	–0.1	–0.7	–0.6	–0.5	–0.1

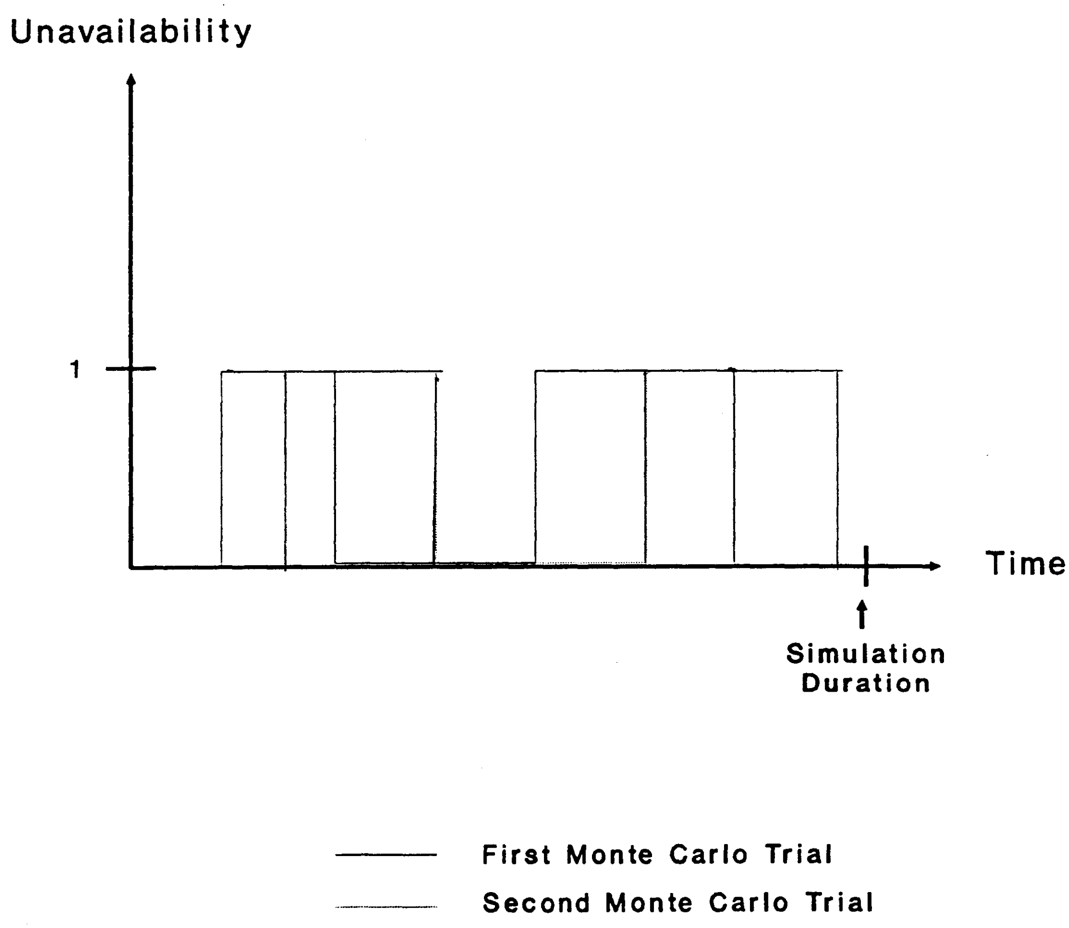


Figure 9. SIMULATION UNAVAILABILITY TIME LINE

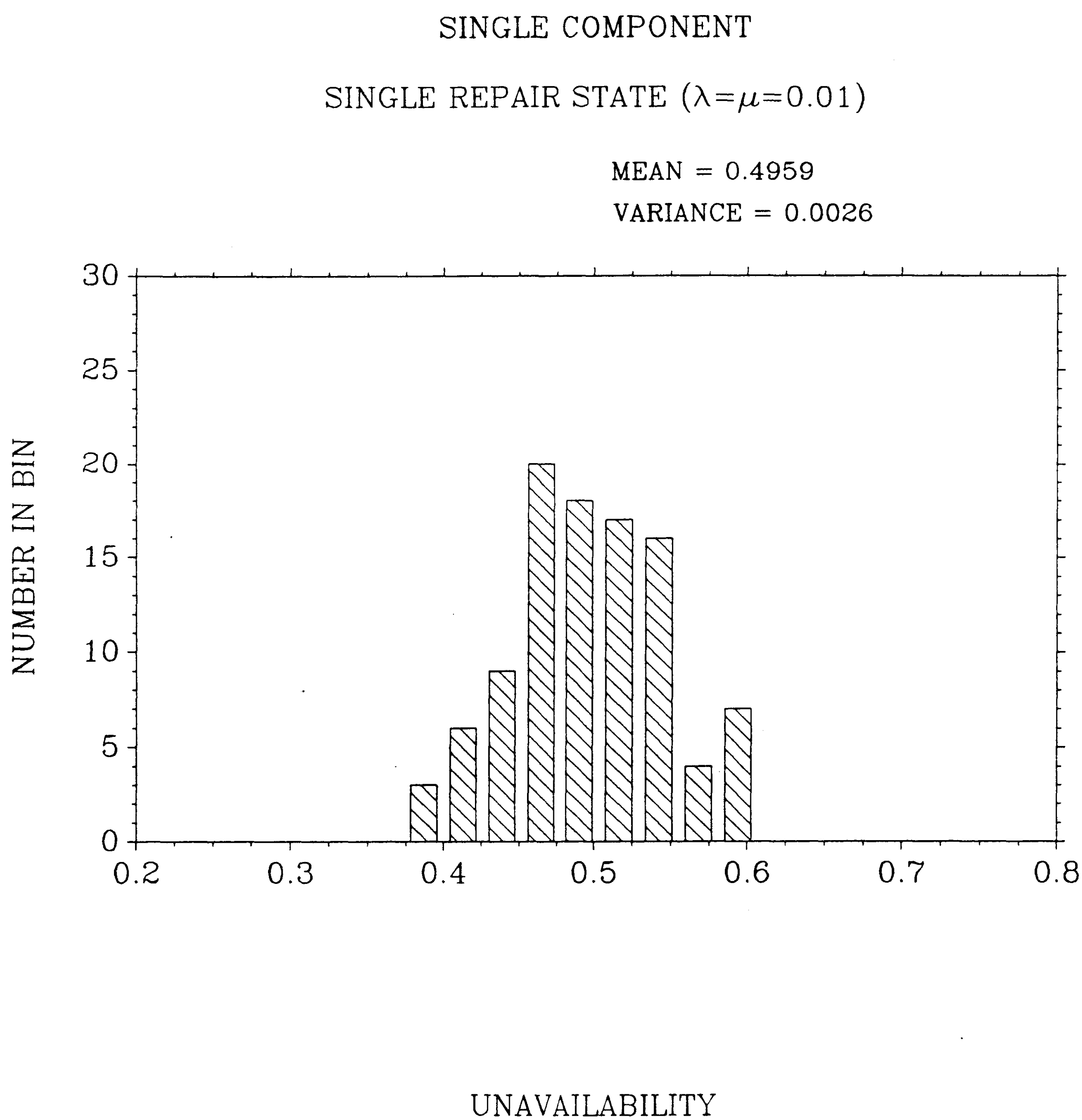


Figure 10. SINGLE COMPONENT, SINGLE REPAIR STATE — AVERAGE UNAVAILABILITY

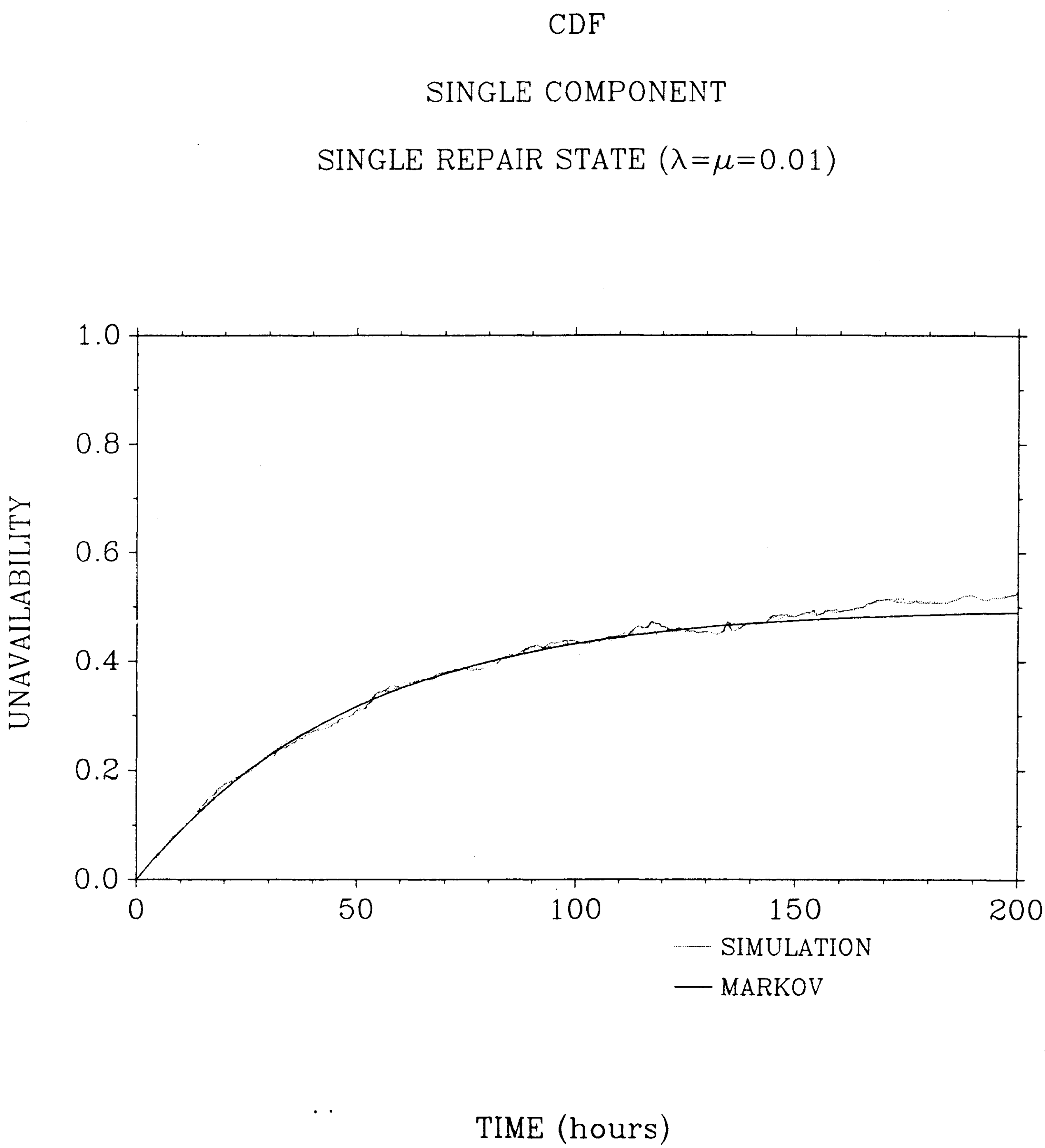


Figure 11. SINGLE COMPONENT, SINGLE REPAIR STATE — TIME DEPENDENT UNAVAILABILITY

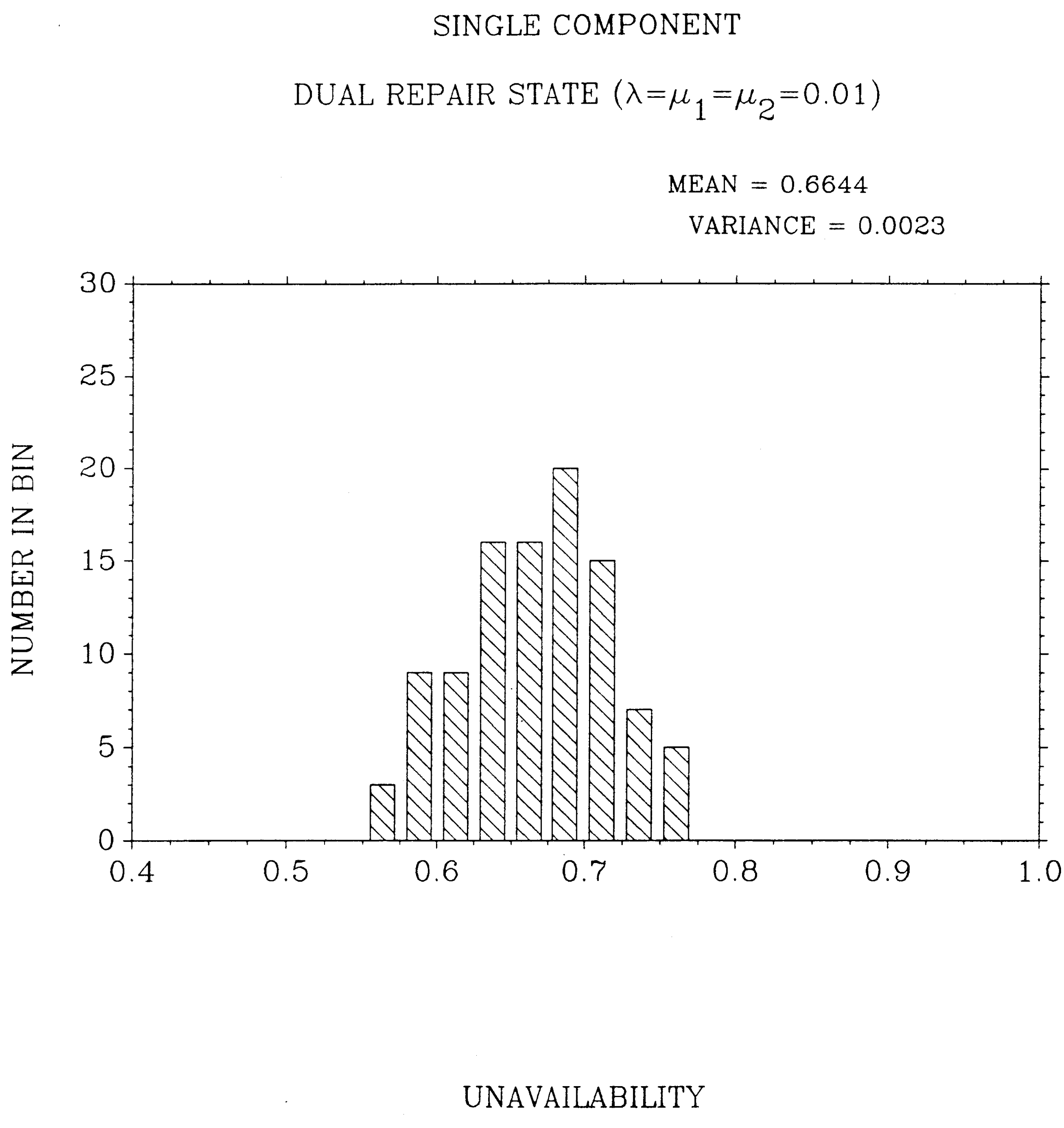


Figure 12. SINGLE COMPONENT, DUAL REPAIR STATE — AVERAGE UNAVAILABILITY

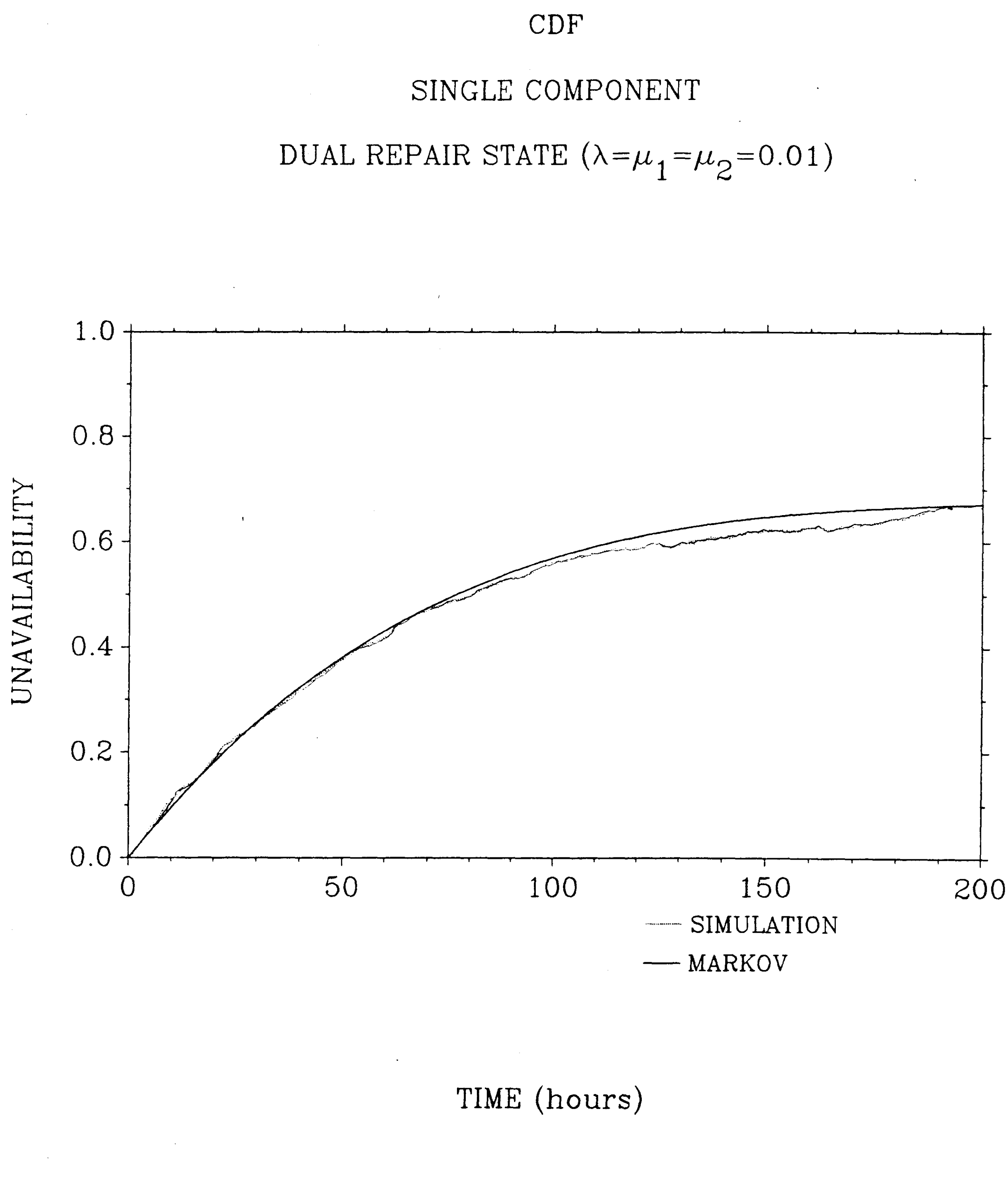


Figure 13. SINGLE COMPONENT, DUAL REPAIR STATE — TIME DEPENDENT UNAVAILABILITY

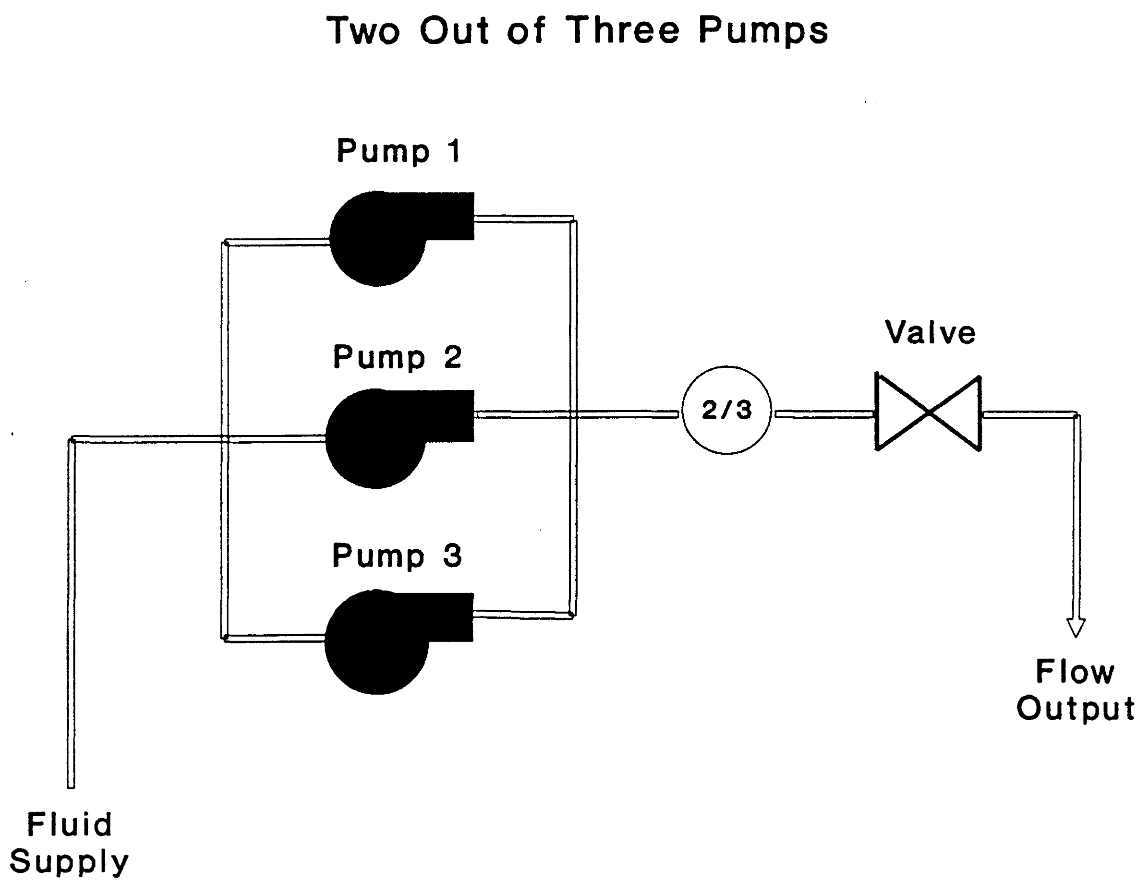


Figure 14. TWO OUT OF THREE PUMPS SYSTEM DIAGRAM

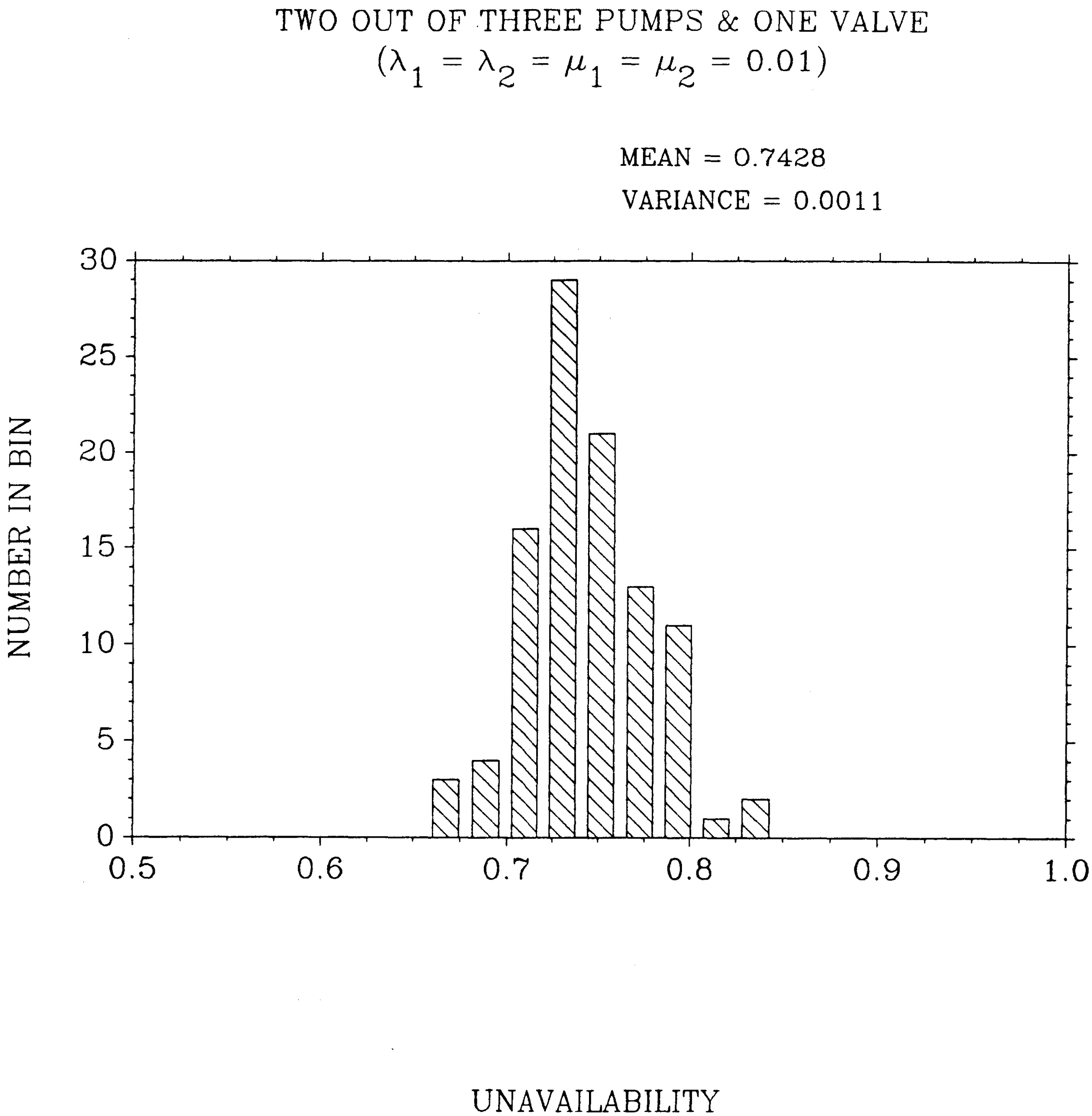
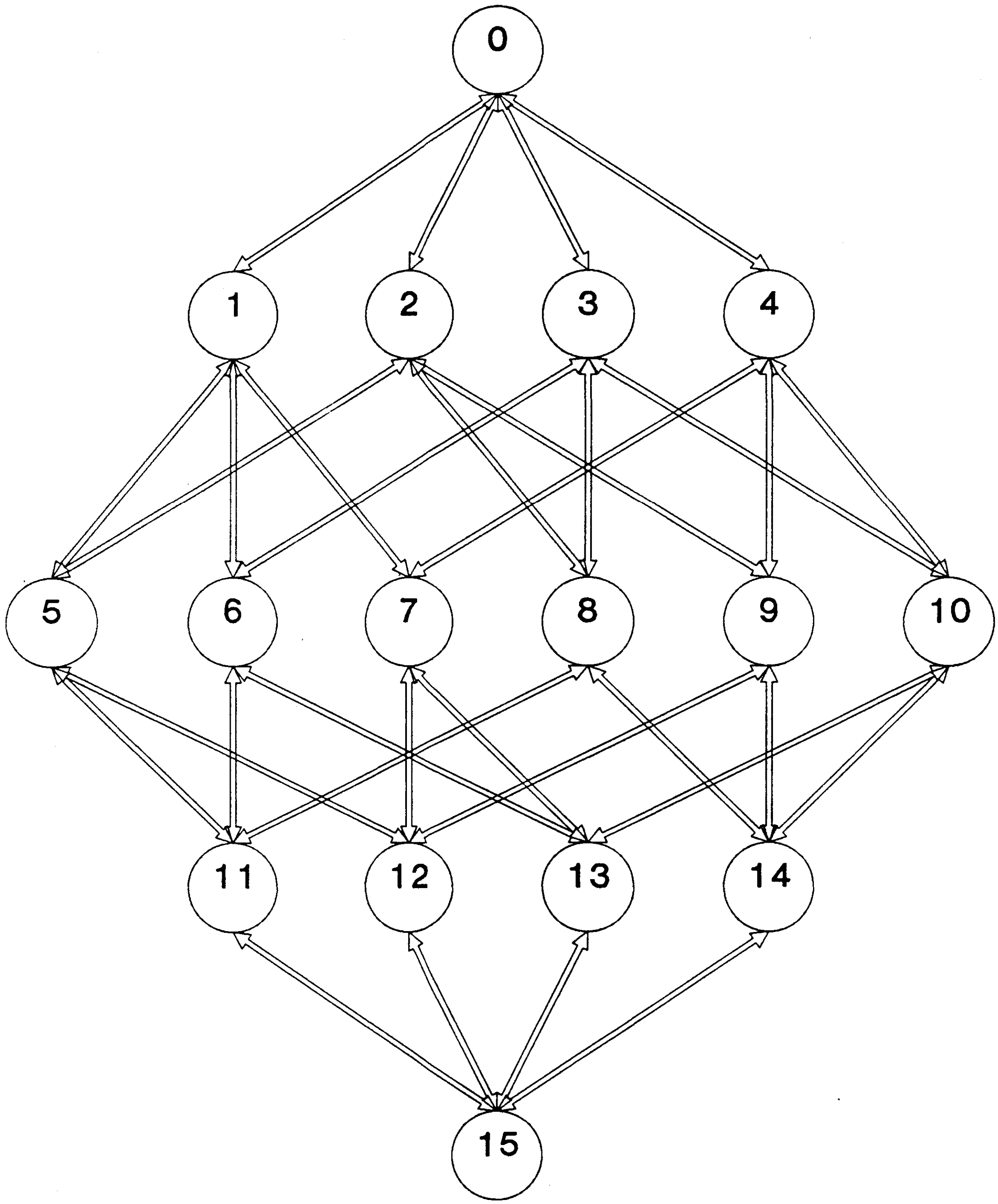


Figure 15. TWO OUT OF THREE COMPONENT — AVERAGE UNAVAILABILTY



Note: All Transfer Rates are Equal

Figure 16. MARKOV STATE TRANSITION DIAGRAM
FOR TWO OUT OF THREE PUMP DIAGRAM

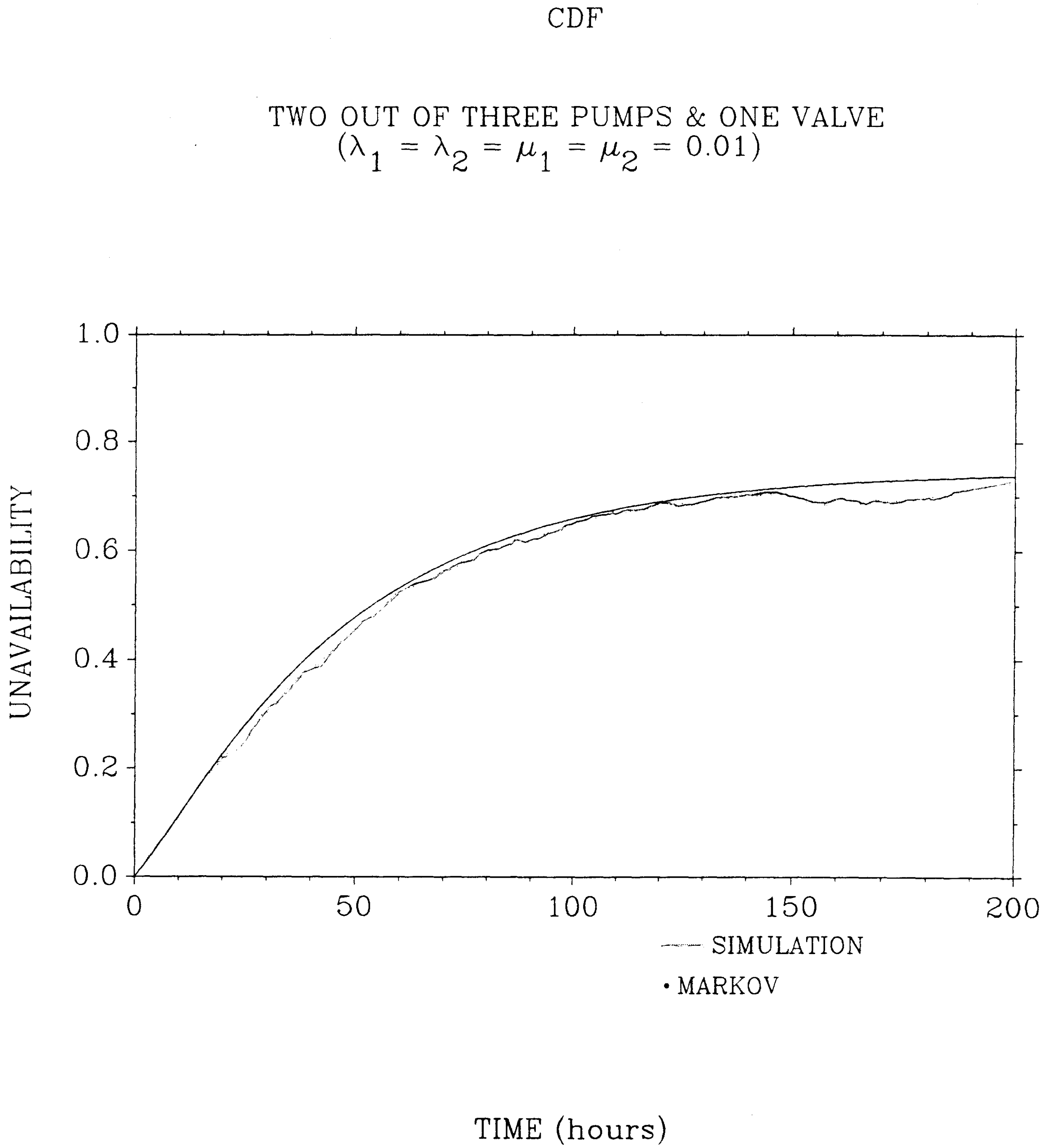


Figure 17. TWO OUT OF THREE COMPONENT — TIME DEPENDENT UNAVAILABILTY

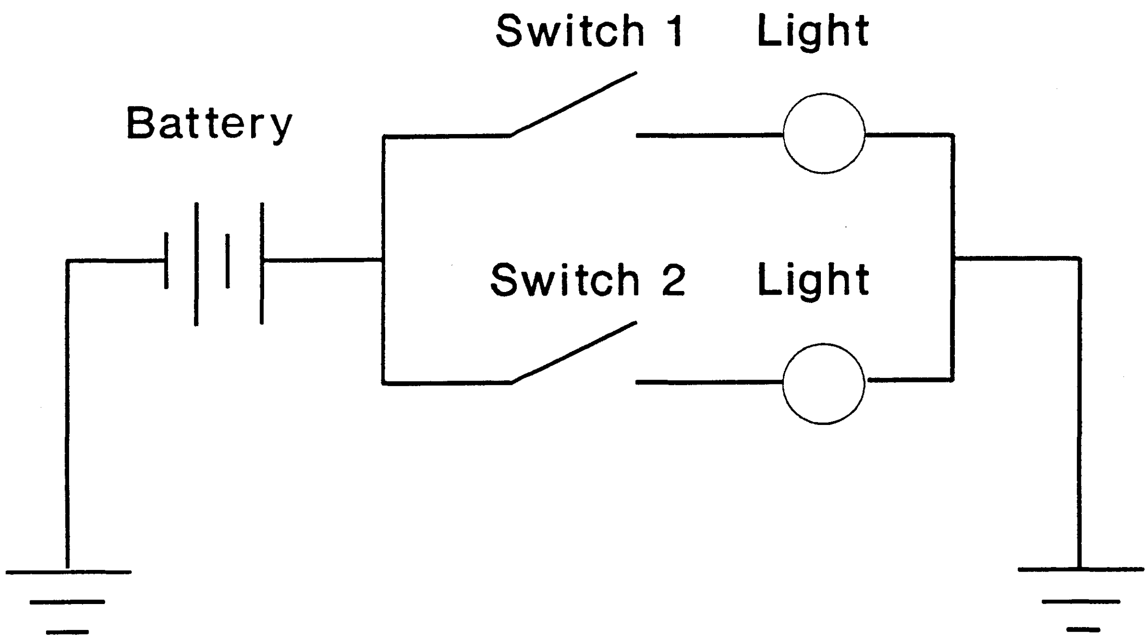


Figure 18. LIGHT BULB PROBLEM DIAGRAM

4. CONTINUOUS SIMULATION APPLICATION

Most reliability analysis methods are designed to treat only systems which can be modeled using a discrete state space. This limitation may be important when analyzing systems whose behavior depends strongly on continuous variables (e.g., pressure, temperature, and flow rates). As an exception to this general rule, Refs. 19 and 20 describe a dynamic methodology based on discrete Markov chains to model process control systems.

The base DYMCAM program discussed in previous sections also does not have the capability to treat failures of components whose state depends on continuous variables. In this section the program is modified to include this capability. The purpose of this analysis is to indicate the magnitude and types of changes required to accomodate this rather large change in problem type.

The particular problem to be solved is similar to the example treated in Ref. 10. The basic problem, and differences between this problem and that described in Ref. 10 are discussed in the following section. The next sections describe the modified program, called TANK, created for the analysis, the results of the simulation, and a simplified Markov chain model used to verify the reasonableness of the results.

4.1 Problem Description

The problem to be solved consists of a fluid containing tank which has three separate level control units. Figure 19 shows a diagram of the system. Each control unit is independent of the others and has a separate level sensor associated with it. The level sensors measure the fluid level in the tank (a continuous process variable) and, based on the information from the level sensors, the operational state of the control units is determined. Each flow control unit can be thought of as containing a controller which turns the unit on and off based on the signal from the level sensors, as shown in Figure 19. Failure of the system occurs when the tank either runs dry or overflows.

The tank has a nominal fluid level at the start of system operation of zero meters. The maximum level of the tank is 3 meters (point b) and the minimum level of the tank is –3 meters (point a). If the tank level moves out of this range, failure of the system has occurred. Within this range there are two set points at –1 meter (set point α_1) and +1 meter (set point α_2). These set points define three control regions for system operation. Region 1 is defined from point a to α_1, Region 2 is from α_1 to α_2, and Region 3 is from α_2 to point b. When the fluid level is in any of the three control region there is a specific action required of each of the three control units. Each control unit acts independently and is not aware of what the state of the other control units is except through the change occurring in the process variable. Table ?? shows the control unit states for each control region.

Unit 1 is an outlet element providing a means for releasing fluid from the tank to lower the level. As in Ref. 10, Unit 1 is assigned an exponential failure distribution with a mean failure time of 320 hours. When operating, the unit allows fluid to flow out of the tank. The associated rate of tank level change is 0.01 meters per minute. Unit 1 receives a command to turn on (open) command from the level controller when the fluid level is in Regions 2 and 3, and it receives a signal to turn off (close) when the fluid level is in Region 1. If the unit is modeled as a valve, it is clear that the valve is normally open unless the fluid level is below the low level setting for the tank, in which case the valve is closed. The component routine used to model Unit 1 as a valve is one of the routines contained in the basic DYMCAM program code.

Unit 2 is a supply unit which provides fluid input to the tank. It too has an exponentially distributed failure time. The mean failure time used is the same as that used in Ref. 10: 219 hours. When operating, the unit supplies fluid, leading to a tank level change rate of 0.01 meters per minute. This unit receives a control signal to turn on (open) if the fluid level is in Regions 1 and 2, and it receives a signal to turn off (close) if the fluid level is in Region 3. (Note that the unit can be modeled as a pump or an inlet valve; the latter is used in this work.)

The third unit is also a fluid supply element. It is identical in nature to Unit 2 except that it has a mean failure time of 175 hours. Two different tank level change rates are associated with Unit 3: 0.01 meters/minute and 0.005 meters/minute. The former corresponds to Case A studied in Ref. 10; the latter corresponds to Case F of Ref. 10. Unit 3 is normally in an off (closed) state unless the fluid level drops into Region 1, in which case the unit receives a signal to turn on (open). Like Unit 2, this unit is modeled as an inlet valve.

At the start of system operation the fluid level is in the normal region (Region 2) and Units 1 and 2 are on while Unit 3 is off. Thus the flow rate into the tank is equal to the flow rate out of the tank, and the fluid level remains constant. This state will continue until one of the level control units fails. Then, the fluid level will change either up or down depending on which unit has failed; when the fluid level enters a new control region the controller will take action to halt the change. The new system state may or may not be stable, as is seen later in the section, however failure of the system cannot occur with the failure of a single control unit. The level will remain in the new control region, or oscillating between two control regions until a second unit fails. The second failure is likely to cause the system to fail by the tank either running dry or overflowing.

Since component repair is not considered in this problem, all scenarios will end in system failure. The type of failure experienced is dependent on the sequence in which the units fail and also upon the timing of failure for certain cases in which the fluid level oscillates. The purpose of this reliability analysis is to determine the time dependent probability of each of the two types of failure. The complication which prohibits this type of problem from being easily solved by other analysis methods is that component states are dependent on a continuous process variable. For exact results, modeling of the process variable must be done, and a method must be available by which control units are allowed to change state at non–deterministic times. In other words the method of the DYMCAM program, which uses external events to control phased mission problems, is not appropriate since the time at which a component will be required to change its operating state will not be known before the simulation is begun.

It should be pointed out that there are differences between the problem treated in this section and that treated in Ref. 10. In Ref. 10, control units are only allowed to be in one of two states: "on" or "off". If a failure occurs, the control unit is assumed to transfer to the wrong state, which depends on the particular control region inhabited by the fluid level. For example, if the tank level is in Region 1 and Unit 1 fails, it is assumed that Unit 1 is "on." Thus, Ref. 10 treats the failures as being failures of the control system. Note that because "failure" is defined in the context of the control region, this means that, in principle, a unit can change states when the tank level moves from one region to another, even if the unit is failed.

By contrast, this work allows four states for the components; they can be "on", "off", "failed on", and "failed off". Once a component fails, it remains in that particular failed state regardless of any changes that may occur in the rest of the system. Thus, the failure model used in this work is more component oriented. Note that if the 4–state component model is used in the discrete Markov chain approach, transitions among the 64 possible hardware states must be considered explicitly (as opposed to the 8 possible hardware states treated in the 2–state model).

4.2 The TANK Program – Modifications to DYMCAM

The major change needed in the DYMCAM program in order to solve the tank problem is to add a routine which models the continuous process variable (tank level).

SIMSCRIPT II.5 has a continuous variable modeling capability, described in Ref. 18, and this is used to treat the tank level. This new variable requires the addition of several subroutines to the DYMCAM program and these are described in this section. In addition, certain subroutines of the original program required minor modification. Table 8 lists all the new subroutines added and all the old subroutines to which adjustments were made. A complete listing of the new subroutines is contained in Appendix C. The modified subroutines are contained in Appendix B. In Appendix B, those subroutines which were modified for the tank problem contain the message "TANK" at the far right hand side of the page next to the added or altered lines of code. These commands should be removed or altered to use the DYMCAM program by itself. It should be emphasized that the sole purpose of the particular modified program is to demonstrate an application of the simulation modeling approach to a reliability problem involving continuous process variables. The modifications made to the DYMCAM program in this demonstration have been chosen with an eye on rapid implementation rather than programming generality.

The most fundamental addition to the program is the TANK process routine. This is the continuous process which provides SIMSCRIPT with the capability to solve continuous variable systems. In pure discrete event simulation, the model advances in time from event to event using entries in an event queue. It is assumed that the system remains unchanged between scheduled events and can change only at the designated event times. For a continuous model, variables are assumed to vary continuously with advancing time. Thus time is incremented by a small amount and all variables are updated. This is done by associating a differential equation with each continuous variable which indicates the rate of change for that variable. Then as time is advanced by discrete time steps, integration is performed to update the status of the continuous variable at the end of each time step. (Of course, the "continuous updating" of variables can be viewed as the deterministic scheduling of events over relatively short time intervals.)

SIMSCRIPT II.5 allows the use of a variable time step for which the user must specify the minimum and maximum values. The integration routine can be specified explicitly, or the Runge–Kutta integration routine which is contained in the SIMSCRIPT language may be used. Also associated with the integration routine are error parameters that must be provided to specify the accuracy of integration calculations desired. All of these initializations are entered in the TANK.INITIALIZE.RUN routine.

Figure 20 shows a flow chart of the operation of the TANK program. Following through this chart will provide an explanation of the TANK program operation and methodology. The function of the base DYMCAM routines are described in Section 2.

The analysis begins with the TANK.INITIALIZE.RUN routine which creates and initializes the variables and signals associated with the tank. This is done only once at the beginning of each computer run. Next, for every trial, the tank output signals, the tank level, and the initial flow rate are reset by the TANK.INITIALIZE.TRIAL routine. After all other initialization is completed by the DYMCAM program, the simulation clock is started. Failure of all three units will be scheduled to occur at discrete times in the simulation based on their failure rates, and these times are assigned as in DYMCAM.

Unlike DYMCAM, which uses only discrete event simulation, the TANK program also contains the continuous tank level variable. Thus after the start of the simulation, control of the time aspect of the program is performed by the TANK process. This subroutine contains the statement (Line 15):

```
work continuously evaluating 'water.level' testing 'tank.condition'
```

This statement updates the tank water level using the WATER.LEVEL routine which applies the SIMSCRIPT formulation of the simple differential equation governing the tank level:

```
d.level(tank) = net.flow.rate(tank)
```

The time step used in the TANK program is fixed at one hour. If a variable time step were allowed, then SIMSCRIPT would adjust the step based on how fast the variable is changing. The integration routine, Runge–Kutta in this case, calculates the water level at the new time.

Once the new level is determined, the TANK.CONDITION routine is called to verify that the tank condition is "good." If it is, then the simulation clock is advanced another time step, and the new water level is calculated. If the TANK.CONDITION routine determines that: 1) the net.flow.rate(tank) does not equal the flow.rate.in minus the flow.rate.out, 2) the tank has failed by overflow or dryout, or 3) the control state is not correct based on the current fluid level; then continuous time steps are stopped and control continues in the TANK process. The net flow rate for the tank is then updated. The reason for this is to provide proper synchronization for changing of the flow rate. After updating the net flow rate, the TANK process calls the TANK.UPDATE routine.

The TANK.UPDATE routine serves two functions. First it checks the water level to see if overflow or dryout has occurred. If either condition has occurred, then the output signal from the tank, indicating tank status, is set equal to zero (representing tank failure), and control is returned to the TANK process. The TANK process then suspends itself. The rest of the simulation time of the trial passes in discrete event fashion. When the scheduled STOP.TANK and STOP.SIMULATION times are reached, the TANK process is reset and the next trial begun.

It should be noted that the system indicator variable can have only one of two values indicating either system success or failure. Since both tank overflow and tank dryout are failure events, it is necessary to simulate failure in each mode separately. This is done by altering the computer code to count only failures of one type or the other during a particular run of the program. To test for the probability of tank overflow, Lines 13 through 17 of the TANK.UPDATE routine were rendered un–executable, and when testing for tank dryout, Lines 13 to 17 were restored and Lines 24 through 28 of the TANK.UPDATE routine were removed. In either case, once the tank has run dry or overflowed, continuous operation of the system is suspended. Of course, an alternate modification is to revise the SYSTEM.UPDATE and RUN.OUTPUT routines such that multiple output states are recognized. This was felt to be more complex than the method adapted.

If the tank has not failed, then the TANK.UPDATE routine checks to see if the unit control states are correct based on the fluid level of the tank. If not, the TANK.UPDATE routine creates the proper control signals to send to the three units to change their operating state to the proper condition. To cause the units to change state, the SYSTEM.UPDATE routine is called. This is a DYMCAM routine which changes the states of components based on changes in signals and on changes in other system component states. A new line added to the SYSTEM.UPDATE routine for the TANK problem, appears at Line 141. This command causes the FLOW.UPDATE routine to be called. This routine calculates the flow rate going into the tank and the flow rate coming out of the tank based on the state of the three control units. It does not directly calculate the net flow rate into the tank which is used by the WATER.LEVEL routine. This is done in the TANK process to prevent the flow rate from changing during an integration time step.

Once the flow rates are updated, control is returned to the SYSTEM.UPDATE routine. The SYSTEM.UPDATE routine, in turn, returns control to the TANK.UPDATE routine. Now the tank is in the proper operating condition and thus control is returned to the TANK process. Since the tank has not yet overflowed or run dry, the TANK process begins execution of the continuous function again. Time is advanced by the given time step (one hour), the level of the tank is updated, and the condition of the tank is again checked. As long as the tank condition is good, operation continues in this fashion. If the tank condition tests bad, then the continuous operation is again suspended.

The failure rates used for the three control units in the tank problem make it highly likely that the system will fail during the simulated 1,000 hour time period, therefore at some point the continuous process should stop and the simulation will continue in the discrete event fashion. In the rare case of no system failure during the 1,000 hour period, the continuous process will be suspended by the STOP.TANK routine at the 1,000 hour time point, and the system will be reset for the next trial. Of course, no failure event would be recorded for such a trial.

Individual control unit failures are controlled by the DYMCAM program. When a failure occurs, the SYSTEM.UPDATE routine is called which in turn will cause the flow rate into and out of the tank to be adjusted. This change will affect the TANK program when the TANK.CONDITION routine detects that the net flow rate to the tank does not equal the flow rate in minus the net flow rate out, and as described above, the continuous operation will be interrupted while the net flow rate is changed by the TANK process.

The new routines, TANK.INITIALIZE.RUN and TANK.INITIALIZE.TRIAL areused to initialize all the parameters associated with the test. Most importantly the TANK.INITIALIZE.RUN routine creates all of the output signals associated with the tank. Since the DYMCAM program does not recognize the tank as being a component, it is not assigned any output signals. Thus one line is added to the DYMCAM RUN.INITIALIZE routine (Line 51) to add five signals to the total system signal count. Figure 21 shows all of the signals associated with the TANK program. The five new signals are indicated by stars. These signals are then initialized by the TANK.INITIALIZE.RUN routine. Once created, the signals are treated in the same manner as all other component signals. The five signals concerned are the three control signals from the tank to each of the three units, the output process flow from the tank to Unit 1, and a system status signal to indicate system success or failure.

The TANK.INITIALIZE.RUN routine also creates the signal and component files necessary for clean operation of the program code. The TANK.INITIALIZE.TRIAL routine, which is executed prior to each trial, resets the net flow rate to zero, sets the tank fluid level back to zero, turns the flow out of the tank on, resets the system success indicator to "good," and turns off the command signals to all three control units.

The STOP.TANK process operates in much the same fashion as the STOP.SCENARIO process. It is used to suspend operation of the tank, if the tank has not failed during the simulated time period (which has a very low probability of occurrence), and then to reset the tank so it is ready to be started at the beginning of the next Monte Carlo trial.

Minor modifications were also made to the MAIN routine and the CALL.UPDATE process of the DYMCAM program. The MAIN routine was modified to include calling the tank initialization routines and to call the STOP.TANK process. In addition the availability data structure was modified to print out the desired results in the output file. The CALL.UPDATE process was revised to include Lines 14 and 15 which simply take the tank out of its suspended state and cause it to start operation at the beginning of every trial.

In addition, a number of new lines were added to the PREAMBLE to reflect all of the new routines, processes, and variables associated with the TANK program. These lines are indicated in the PREAMBLE listing for the DYMCAM program in Appendix B by the marker "TANK" which is placed at the far right hand side of each line of code that was modified or added. The entire TANK program, as a unit, was compiled and kept separate from the DYMCAM program, since subroutines cannot be compiled separately, and the two codes are not used together. They do, however, contain the same basic structure and the TANK program should be viewed as an extension of the DYMCAM program, which remains almost entirely intact in the TANK code.

The input file necessary to run the program is exactly the same format as the input file for the DYMCAM program described in Appendix A. The only point to note is that the three units were modeled as valves in the simulation program. It is also important that the names of the level control units be entered as unit1, unit2, and unit3 so that they are recognized by the TANK program as the flow control units. An example input file for this program is contained in Appendix D. The same input file is used for all tests, and changes are made in the program to reflect testing for the failure condition of overflow or dryout and to alter the flow rate provided by Unit 3. The output file generated by the TANK program is identical in format to the output generated by the DYMCAM program, and an example print out is shown in Appendix E.

4.3 Simplified Model for Benchmarking

Because of the differences between the problem analyzed and that treated in Ref. 10, it is expected that there will be some difference in results. To benchmark the TANK computations, therefore, a simplified model for the system is created. This model is based on a comparison of the time scales for component failure and for tank level change. The three control units have mean failure times of 320, 219, and 175 hours respectively. On the other hand, if the tank fluid level is at zero when a unit fails, then at a level change rate of 0.01 meters per minute it will only take approximately 1.7 hours for the tank to change control regions. If the level is at the edge of Region 1, and must travel to Region 3, the longest amount of time that will be required is approximately 3.5 hours. These times are small compared to the mean failure times. In the simplified approach, it is assumed that after one failure occurs, a second failure does not occur until the system has entered a new control region. This allows the treatment of the system using a Markov chain.

The two cases considered correspond to Cases A and F described in Ref. 10. The difference between these two cases lies with the flow rate out of Unit 3. In Case A, the associated tank level rate of change is 0.01 meters/minute. In Case F, the rate of change is 0.005 meters/minute. This difference leads to different sets of potential accident sequences, as discussed below.

4.3.1 Analysis of Case A

For Case A, the tank starts at time zero with all units operational (Units 1 and 2 are turned on, and Unit 3 is turned off). The tank will continue in this state with no change in the tank level until a failure of a control unit occurs. The sequencing of failure is very important so each unit failing first will be considered separately. Figure 22 shows the state transition diagram for this system. All states are defined in Table 9.

The three possible initiating events are Unit 1 or Unit 2 failing closed, or Unit 3 failing open. It can be easily shown that the probability of each individual unit being the first to fail is given simply by the ratio of the failure rate for that unit divided by the sum of the failure rates for all three units. To show this, consider the system composed of only the first four states of Figure 22, states 0, 1, 2, and 3. The four state probability equations for this system are:

$$\begin{aligned}
\frac{dP_0}{dt} &= -(\lambda_1 + \lambda_2 + \lambda_3)P_0 \\
\frac{dP_1}{dt} &= \lambda_1 P_0 \\
\frac{dP_2}{dt} &= \lambda_2 P_0 \\
\frac{dP_3}{dt} &= \lambda_3 P_0
\end{aligned} \tag{6}$$

Since at t = 0, the system is initially in State 0, the time–dependent state probabilities can be easily found:

$$\begin{aligned}
P_0(t) &= \exp\{-(\lambda_1 + \lambda_2 + \lambda_3)t\} \\
P_i(t) &= \frac{\lambda_i}{\lambda_1 + \lambda_2 + \lambda_3} - \frac{\lambda_i}{\lambda_1 + \lambda_2 + \lambda_3}\exp\{-(\lambda_1 + \lambda_2 + \lambda_3)t\} \\
&\text{for } i = 1,2,3
\end{aligned} \tag{7}$$

For t sufficiently large, it is clear that $P_i(t) \rightarrow \lambda_i/(\lambda_1 + \lambda_2 + \lambda_3)$. Using these results it is found that Unit 3 will fail first 43% of the time, Unit 2 34%, and Unit 1 23% of the time.

The initial failure of Unit 1 is the easiest case to consider since it will always lead eventually to a tank overflow condition, regardless of the relative flow rates provided by the three units. Unit 1 failing closed causes the fluid level to rise until it passes into Region 3, at which time Unit 2 is shut off. The tank remains in this condition until either Unit 2 or Unit 3 fails open, either of which will lead directly to a tank overflow condition.

The initial failure of Unit 2 poses a more interesting problem. With Unit 2 failing closed, the fluid level will drop until it reaches Region 1. Then Unit 1 is closed and Unit 3 is opened. This causes the fluid level to rise until the fluid level is in Region 2 again, at which time Unit 1 is opened and Unit 3 is closed. Thus, the fluid level will continue to oscillate about the low level set point of –1 meters with Units 1 and 3 being alternately turned on and off. In this analysis, the time step duration used in the simulation is one hour. Therefore, for this case, the level of the tank will fluctuate between –0.4 meters and –1.6 meters, spending equal time in each of the two control regions (1 and 2). This is true since while the level is rising, the rate of increase is 0.01 meters per minute, and while the level is falling the rate of level change is also 0.01 meters per minute. Fluctuation occurs between the same two points since time steps were forced to be constant at one hour intervals.

From this state there are four possible events that can occur. While the fluid level is rising, Unit 1 can fail open or Unit 3 can fail closed, or while the fluid level is decreasing Unit 1 can fail closed or Unit 3 can fail open. It is clear that if either unit fails while the level is rising the flow rates in and out of the tank will then be equal and the fluid level will stop changing until the failure of the third level control unit. This third failure will lead directly to the tank running dry.

If one of the two control units fails while the tank level is dropping then, again, the tank fluid level will cease to change until the failure of the third unit. This time, the third unit failing will lead directly to overflow of the tank. Since the tank spends an equal time in the rising and falling level states, it is equally likely that the tank will fail in an overflow or dryout state. Thus for the case of unit two being the initial failure event, there is a 50% probability that the tank will fail in each of its two failure conditions.

For the case of Unit 3 failing first, the solution is as easy as for Unit 1 failing first. When Unit 3 fails open, the fluid level will begin to rise until the tank level reaches control region 3, at which time Unit 2 will be closed. Now with both Units 1 and 3 open, the fluid level will hold constant at 1 meter. The next failure event, either Unit 1 failing closed or Unit 2 failing open, will lead directly to a tank overflow condition. Thus for all scenarios where Unit 3 fails first, the tank will fail by overflow.

From the above discussion it is evident that all Unit 1 initial failures, all Unit 3 initial failures, and half of the Unit 2 initial failures will eventually lead to an overflow condition. Thus, using the values quoted above for the probability that each of the three units will fail first, it is found that the probability that the tank will fail by overflow is:

$0.23 + 0.43 + (0.5 * 0.34) = 0.83$

The tank will fail by overflowing approximately 83% of the time and fail by running dry the other 17% of the time.

It is important to note that although the above method simplifies the problem so that it may be solved with Markov chains without even considering the continuously variable tank fluid level, this method is only an approximation and is as good as the assumption that two failures do not occur within a 3.5 hour time period. This, of course, will not be the case for all continuous variable process control problems. In this example problem the results obtained using the approximation agree well with the simulation results, but several possible failure sequences which will occur with low probability are ignored. For example, consider the case of failure of both Units 2 and 3 within 1.5 hours of each other. This will leave the fluid level essentially unchanged or, at least, still in control region 2. The net flow rate from the tank is still zero so the tank will remain in this condition until Unit 1 fails, at which time the tank will overflow. If it is considered that Unit 3 fails just prior to Unit 2, then the result is consistent with the approximate analysis. However if Unit 2 failed first, then the approximate method predicts that half the cases will experience system failure by overflow and half will be by dryout. This is obviously not the case for the dual failure example and the approximate solution will be slightly in error. Other "simultaneous" failures lead to similar conclusions.

4.3.2 <u>Analysis of Case F</u>

For Case F the problem becomes much more complicated. The initial failure probabilities remain unchanged from Case A, but some of the sequences of events after initial failure change. One part that remains the same, however, is the scenario following initial failure of Unit 1. Since Unit 1 is the only way fluid can be removed from the tank, once it has failed closed the tank is guaranteed to fail by overflow. Thus, as in Case A, if Unit 1 fails first, all scenarios lead to overflow. The time to overflow, however, could be different due to the different flow rate from Unit 3.

If Unit 2 fails first, the tank level drops to the low set point and begins to oscillate above and below this mark as Units 1 and 3 are opened and closed (as in Case A). However, the amount of time spent in each control region will be different. When the fluid level is rising, Unit 1 is closed and Unit 3 is open, thus the level is changing at the rate of

0.005 meters per minute. When the level is falling, Unit 1 is open and Unit 3 is closed, thus the level is changing at 0.01 meters per minute. Define the level change rates associated with the flow from each of the three units as x_1, x_2, and x_3 respectively. For Case F the normal values are, $x_1 = 0.01$, $x_2 = 0.01$, and $x_3 = 0.005$ meters per minute. Since Unit 2 has failed closed, then $x_2 = 0.0$. Define the net flow rate as x_{net}, then while the water level is in control region 1 (and Unit 1 is closed), x_{net} is given by:

$$x_{net} = x_3 = 0.005$$

While the water level is in control region 2 (and Unit 3 is closed), x_{net} is given by:

$$x_{net} = -x_1 = -0.01$$

Therefore, if the tank level is considered to vary between the same two levels, the tank must spend twice as much time in the control region one (with Unit 3 open and Unit 1 closed), than in the control region 2 (with Unit 1 open and Unit 3 closed). This is reflected in the failure scenarios.

If, while the tank level is increasing, either unit fails, then the tank will immediately run dry. This is the same result as for Case A except that Case A would not experience dryout until all three units have failed. If while the tank level is decreasing, Unit 1 fails, then the tank level will hold constant until Unit 3 fails open. Then the tank will overflow. This sequence is the same as for Case A; however overflow will occur a few hours later due to the slower flow rate from Unit 3.

The fourth possible failure sequence resulting from the initial failure of Unit 2 is entirely different. If Unit 3 fails while the tank level is decreasing, then the level will continue to decrease until the level reaches control region 1, since the flow through Unit 3 is half the value of the flow through Unit 1. Once in control region 1, Unit 1 is closed and the level will rise because of the flow from failed Unit 3. Once the level is again in control region 2, Unit 1 will be opened. Thus the level oscillates about the –1 meter level with equal time spent while the tank level is rising and falling due to the fact that the flow rate from Unit 1 is exactly twice that from Unit 3 (so the net rates at which the tank level rises and falls are equal).

From this condition, Unit 1 can either fail open or closed depending on whether it fails while the tank level is rising or falling. These failures occur with equal probability. Therefore, once Units 2 and 3 have failed, there is an equal chance that the tank will run dry or overflow.

Summarizing the possible sequences following failure of Unit 2, it is seen that the probability of subsequent failure of Unit 1 or 3 is equal to the ratio of their failure rates to the sum of the failure rates. Thus there is a 65% chance that the next failure will be of Unit 3 and a 35% chance that the next failure will be of Unit 1. Of these percentages, two thirds of the Unit 1 failures will be Unit 1e failing open, which leads directly to dryout, and the other one third of the Unit 1 failures lead to eventual tank overflow. For the Unit 3 failure cases, two thirds will be Unit 3 failing closed, while the fluid level is rising, and this leads to the tank failing by dryout. The other one third lead to oscillation in the fluid level with Unit 1 opening and closing; thus, 50% will lead to eventual system overflow and 50% will lead to system dryout. Evaluating the probabilities of the scenarios initiated by the failure of Unit 2, it is found that 77% lead to tank dryout while 23% lead to tank overflow. Figure 23 shows the state transition diagram for the Case F tank problem.

In Case F it is also no longer true that the initial failure of Unit 3 will eventually lead to tank overflow. To see this, the scenarios associated with the initial failure of Unit 3 are analyzed. Following failure of Unit 3 the tank level rises into control region 3 and then Unit 2 is closed. Since the flow rate from Unit 1 is greater than the flow rate of Unit 3, the level drops into control region 2, at which point Unit 2 is turned back on. Thus the fluid level oscillates about the +1 meter level with Unit 2 being opened and closed. While Unit 2 is on, the net flow rate into the tank is 0.005 meters per minute, and while Unit 2 is off the flow rate out of the tank is 0.005 meters per minute. Thus, if the tank level is assumed to oscillate between the same two levels, the system spends equal time with Unit 2 open or closed.

The next failure of either Unit 1 or 2 will again be in proportion to the failure rates associated with each unit. Using these values it is found that subsequent to failure of Unit 3, there is a 41% chance that the next failure will be of Unit 1 and a 59% chance that the next failure will be of Unit 2. If Unit 1 fails, it closes, and the tank will go immediately to the overflow condition. Since Unit 2 spends fifty percent of its time open and fifty percent of its time closed, it has an equal probability of failing either closed or open.

If while the tank level is decreasing, Unit 2 fails on, the tank will go directly to an overflow state. If, however, Unit 2 fails closed while the tank level is increasing, then the tank level will fall until it is in control region 1, at which time Unit 1 will be closed. Then the level will rise due to flow from Unit 3 until the level is in control region 2, when Unit 1 will be opened again. Thus the level oscillates about the –1 meter level with Unit 1 opening and closing.

The magnitude of the tank level rate of change when the tank level is dropping is the same as when the fluid level is rising, therefore Unit 1 spends an equal amount of time open and closed. If Unit 1 fails closed while it is open, then the tank will overflow. If Unit 1 fails open while it is closed, then the tank will run dry. The latter case was not possible in Case A.

Summarizing the scenarios following the initial failure of Unit 3 it is seen that all but one of the situations leads to a tank overflow condition. If Unit 1 fails second, then overflow is certain to occur while if Unit 2 fails second only three quarters of the time will overflow occur. Evaluating numerically, following the initial failure of Unit 3, there is a 85% chance that the tank will fail by overflow and only a 15% chance that the tank will run dry.

Compiling the results of all initial failure events and evaluating the numerical results it is found that for Case F, the probability that the tank will fail by running dry is 0.30 and the probability that the tank will fail by overflowing is 0.70. Thus in Case F the tank is more likely to fail by running dry than in Case A due to the decreased flow rate from Unit 3. Table 10 summarizes the possible failure sequences for Case A, their probability of occurrence, and the end result. Table 11 summarizes the same results for Case F.

4.3.3 Markov Model

From the above scenario analyses, and making the assumption that the time required for the fluid level to transit between control regions is negligible, it is possible to construct Markov chains to approximate the time dependent behavior of the system. Figure 22 shows the Markov state transition diagram for Case A and indicates that sixteen states are required. Figure 23 shows the state transition diagram for Case F, which requires nineteen states. Table 9 shows the states used for Case A and their corresponding definition. States 11 and 13 correspond to tank dryout while States 4, 5, 10, 12, 14, and 15 correspond to tank overflow. For Case F, there are nineteen states of interest. These states are listed in Table 12. States 6, 12, 13, and 17 contribute to tank dryout while States 4, 5, 10, 14, 15, and 18 contribute to tank failure by overflow.

From the state definitions given in this table, the Markov equations are written in the usual manner. These equations are solved using a 4th order Runge–Kutta scheme. The time period of concern is from time zero up until approximately 1,000 hours. The time dependent results for appropriate states were summed to obtain the time dependent probability of system failure by overflow or by dryout.

4.4 Simulation Analysis

Each TANK program simulation was run for a simulated time duration of 1,000 hours; 1,000 Monte Carlo trials were performed. The Case A results for tank dryout are plotted along with the Markov approximation in Figure 24; the analogous results for overflow are plotted in Figure 25. Both of these figures indicate good agreement between the simulation results and the Markov approximation. The time dependent behavior is virtually identical and values differ by only a few percent. Good agreement between the simulation results and the simplified Markov model is expected since the time required for the tank level to change is small in comparison with the failure times associated with the individual flow control units.

A quantitative comparison of the simulation and simplified Markov results with the numerical results provided by Ref. 9's dynamic Markov approach for Case A is shown in Figure 26. The data for the Ref. 10 curve was provided in Ref. 21, and is the same as the results presented in Ref. 10. The figure indicates that the simplified Markov results agree almost exactly with Ref. 9's predictions and the simulation method provides results which are very similar to both. For this case, the difference between the three methods is very small and indicates that although the approach to the problem was different for each method, the results are quite comparable.

As in the case of Case A, the Case F runs involved a simulated period of 1,000 hours and 1,000 trials. Results of the simulation program are plotted in Figures 27 and 28 along with the Markov predictions for comparison. The results indicate reasonable agreement between the two methods. However, these results do not agree as closely as those obtained from Ref. 21. This is believed to be due to the different treatments of component failures used. These differences are observed in Case F because the different flow rates for the control units lead to different failure scenarios. Note that the data plotted for the dynamic Markov model are obtained from Ref. 21 and are corrected versions of the data presented in Ref. 10.

It is interesting to observe that in the Monte Carlo trials, it was found that 13 of the 1,000 trials involved failure of two units during the same continuous process integration time step (i.e., the failures occurred within one hour). Thus approximately 1.3 percent of the time the assumption made for the initiating event Markov analysis is not valid.

The amount of computer time required to run the simulation is significant. Using an integration time step of one hour, the program takes two hours and fifty minutes for the Case A problem. Using the same parameters with the Case F problem, the test takes four hours and forty–three minutes. The time for Case F is much longer because in this case, there are many more instances where the level of the tank oscillates about either the low or the high tank level set points. The time stated above is for runs on a COMPAQ 386SX personnel computer; times on an IBM XT are estimated to be about six times as long. Thus the time requirement for using this simulation method may be prohibitive.

Certain improvements may be possible to reduce the computer time required. One of these is to increase the length of the integration time step used by the continuous process routine. Another is to more efficiently code the portions of the model which lead to oscillation of a component. Based on the difference in time required for Case A and Case F, this improvement alone may reduce solution time by 75% or more. Other techniques for optimizing the computer code may also certainly be possible, as discussed in Section 5.

4.5 Summary

In this section, the use of continuous simulation methods is explored; these methods are useful for analyzing the reliability of complex process control systems. The specific problem investigated is the tank level control problem addressed in Ref. 10. The simulation solution proposed is a modified version of the DYMCAM program discussed in previous chapters. This new program, called TANK, makes use of the continuous capability available in the SIMSCRIPT II.5 simulation language.

The TANK program is constructed from the previously discussed DYMCAM program; most of the latter was left intact with only minor changes being made to a few lines of the SIMSCRIPT code. Several routines were added to define the continuous variable to be used in the simulation. The key new routine is the TANK process, which models the fluid level as a continuous variable, monitors the level to determine the control region the system is in, and based on this information, causes the opening and closing of control valves.

The TANK program was run for a simulated time period of 1,000 hours and for 1,000 Monte Carlo trials to estimate the time dependent probability of dryout and overflow. The results of the simulation compare well with those from an approximate Markov chain approach, and from a more general Markov model described in Ref. 10.

The computer time requirements for running the TANK program on a personal computer are quite large. This is due in large part to the presence of the oscillation of the fluid level about the upper or lower tank level set points. To reduce computer time requirements, it is possible to revise the code to reflect a more efficient program, and the integration time step can be increased. To increase the accuracy of the results, a larger number of trials must be performed. Since the time required is directly related to the number of trials performed, variance reduction techniques will probably be required.

Table 7

FLOW CONTROL UNIT STATES AS A FUNCTION OF FLUID LEVEL

Control Region	Liquid Level (x)	Control Unit State: Unit 1	Unit 2	Unit 3
1	$x<\alpha_1$	off	on	on
2	$\alpha_1<x<\alpha_2$	on	on	off
3	$\alpha_2<x$	on	off	off

Table 8

TANK SUBROUTINES

Subroutine	Description
<u>Modified DYMCAM Routines</u>	
PREAMBLE	Modified to reflect new variables and processes
MAIN	Modified to initialize and stop the tank
CALL.UPDATE	Modified to start tank process
RUN.INITIALIZE	Modified to add signals
SYSTEM.UPDATE	Modified to update flow rates
<u>New Routines</u>	
FLOW.UPDATE	Routine to calculate flow to and from Tank
STOP.TANK	Process to reset tank after each trial
TANK	Continuous process to monitor fluid level
TANK.CONDITION	Function that checks for proper control region operation
TANK.INITIALIZE.RUN	Routine to initialize all variables and sets for the Tank
TANK.INITIALIZE.TRIAL	Routine to re–initialize specific variables for next trial
TANK.UPDATE	Routine to track System status and control all units
WATER.LEVEL	Routine providing integration quantity for continuous routine

Table 9

MARKOV STATES FOR TANK CASE A

STATE	FAILURE DESCRIPTION
0	All units good
1	Unit 1 failed closed
2	Unit 2 failed closed
3	Unit 3 failed open
4	Unit 1 failed closed then Unit 2 failed open (Overflow)
5	Unit 1 failed closed then Unit 3 failed open (Overflow)
6	Unit 2 failed closed then Unit 1 failed closed
7	Unit 2 failed closed then Unit 1 failed open
8	Unit 2 failed closed then Unit 3 failed open
9	Unit 2 failed closed then Unit 3 failed closed
10	Unit 2 failed closed then Unit 1 failed closed then Unit 3 failed open (Overflow)
11	Unit 2 failed closed then Unit 1 failed open then Unit 3 failed closed (Dryout)
12	Unit 2 failed closed then Unit 3 failed open then Unit 1 failed closed (Overflow)
13	Unit 2 failed closed then Unit 3 failed closed then Unit 1 failed open (Dryout)
14	Unit 3 failed open then Unit 1 failed closed (Overflow)
15	Unit 3 failed open then Unit 2 failed open (Overflow)

Table 10

CASE A FAILURE SEQUENCE SUMMARY

Failure Sequence	Probability	Result
#1 closed, #2 open	0.10	overflow
#1 closed, #3 open	0.13	overflow
#2 closed, #1 closed, #3 open	0.06	overflow
#2 closed, #1 open, #3 closed	0.06	dryout
#2 closed, #3 open, #1 closed	0.11	overflow
#2 closed, #3 closed, #1 open	0.11	dryout
#3 open, #1 closed	0.17	overflow
#3 open, #2 open	0.25	overflow

Table 11

CASE F FAILURE SEQUENCE SUMMARY

Failure Sequence	Probability	Result
#1 closed, #2 open	0.10	overflow
#1 closed, #3 open	0.13	overflow
#2 closed, #1 open	0.08	dryout
#2 closed, #1 closed, #3 open	0.04	overflow
2 closed, #3 open, #1 closed	0.04	overflow
#2 closed, #3 open, #1 open	0.04	dryout
#2 closed, #3 closed, #1 open	0.15	dryout
#3 open, #1 closed	0.17	overflow
#3 open, #2 open	0.13	overflow
#3 open, #2 closed, #1 open	0.06	dryout
#3 open, #2 closed, #1 closed	0.06	overflow

Table 12

MARKOV STATES FOR TANK CASE F

STATE	FAILURE DESCRIPTION
0	All units good
1	Unit 1 failed closed
2	Unit 2 failed closed
3	Unit 3 failed open
4	Unit 1 failed closed then Unit 2 failed open (Overflow)
5	Unit 1 failed closed then Unit 3 failed open (Overflow)
6	Unit 2 failed closed then Unit 1 failed open (Dryout)
7	Unit 2 failed closed then Unit 1 failed closed
8	Unit 2 failed closed then Unit 3 failed open
9	Unit 2 failed closed then Unit 3 failed closed
10	Unit 2 failed closed then Unit 1 failed closed then Unit 3 failed open (Overflow)
11	Unit 2 failed closed then Unit 3 failed open then Unit 1 failed closed (Overflow)
12	Unit 2 failed closed then Unit 3 failed open then Unit 1 failed open (Dryout)
13	Unit 2 failed closed then Unit 3 failed closed then Unit 1 failed open (Dryout)
14	Unit 3 failed open then Unit 1 failed closed (Overflow)
15	Unit 3 failed open then Unit 2 failed open (Overflow)
16	Unit 3 failed open then Unit 2 failed closed
17	Unit 3 failed open then Unit 2 failed closed then Unit 1 failed open (Dryout)
18	Unit 3 failed open then Unit 2 failed closed then Unit 1 failed closed (Overflow)

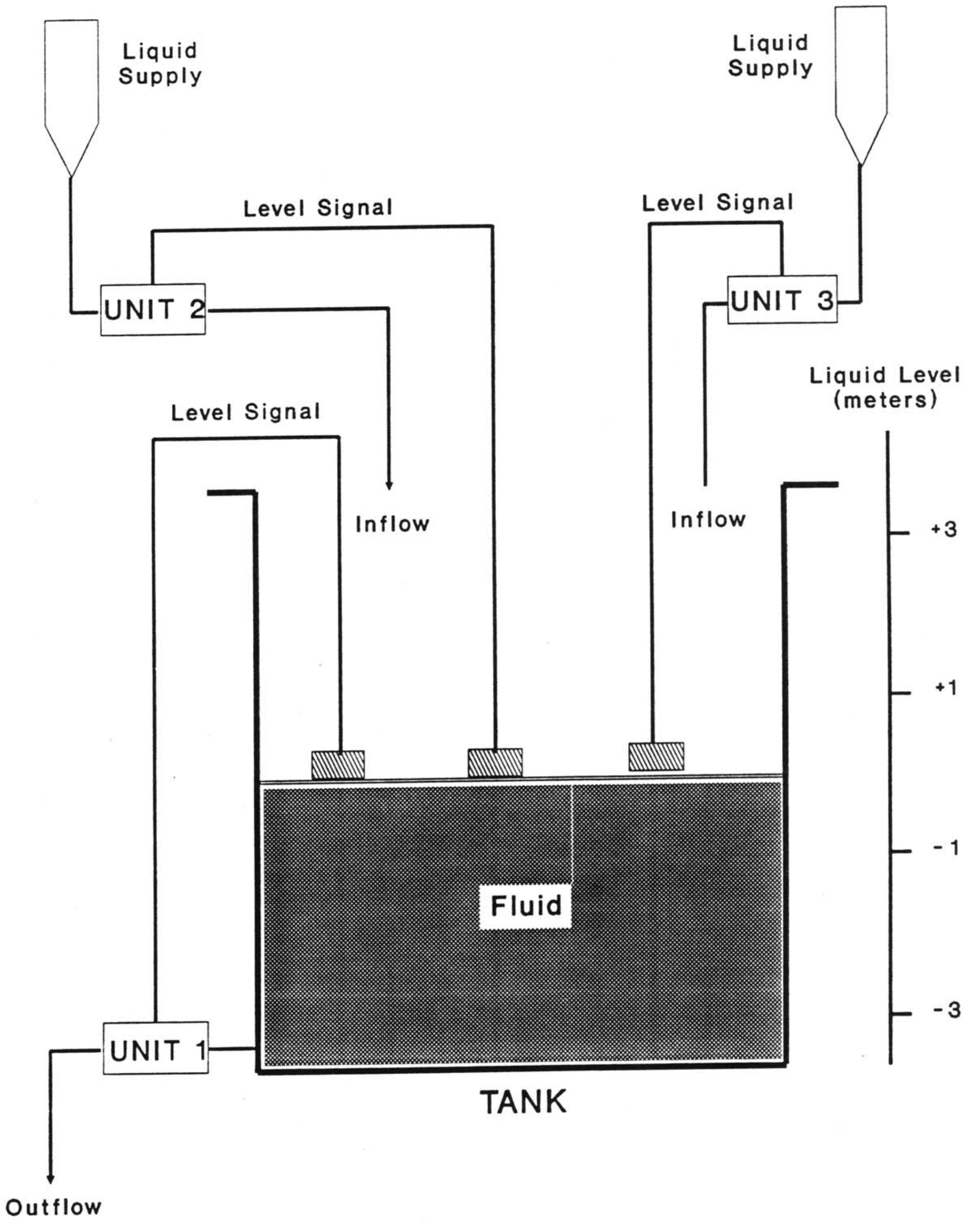


Figure 19. TANK PROBLEM DIAGRAM

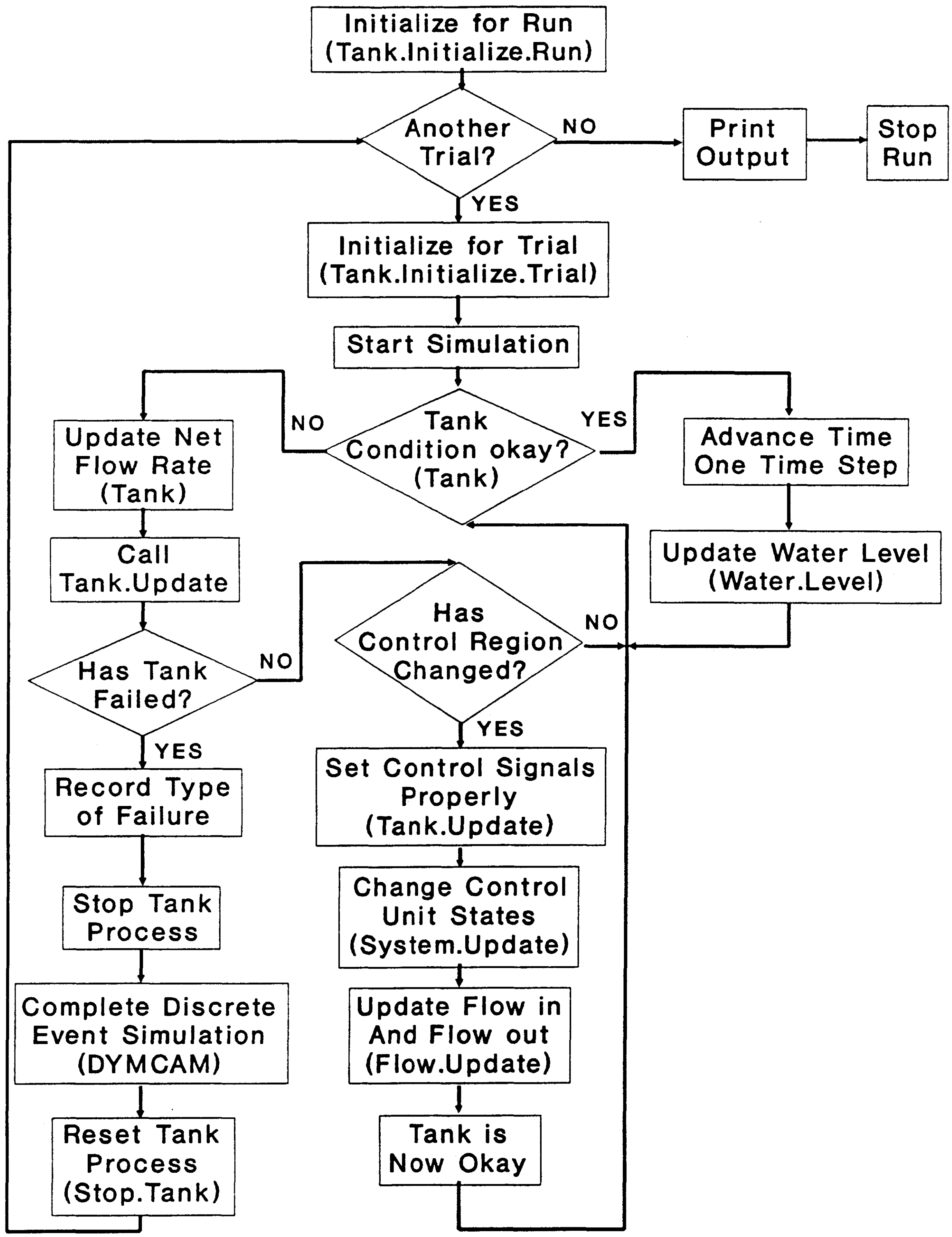


Figure 20. FLOW CHART OF TANK PROBLEM

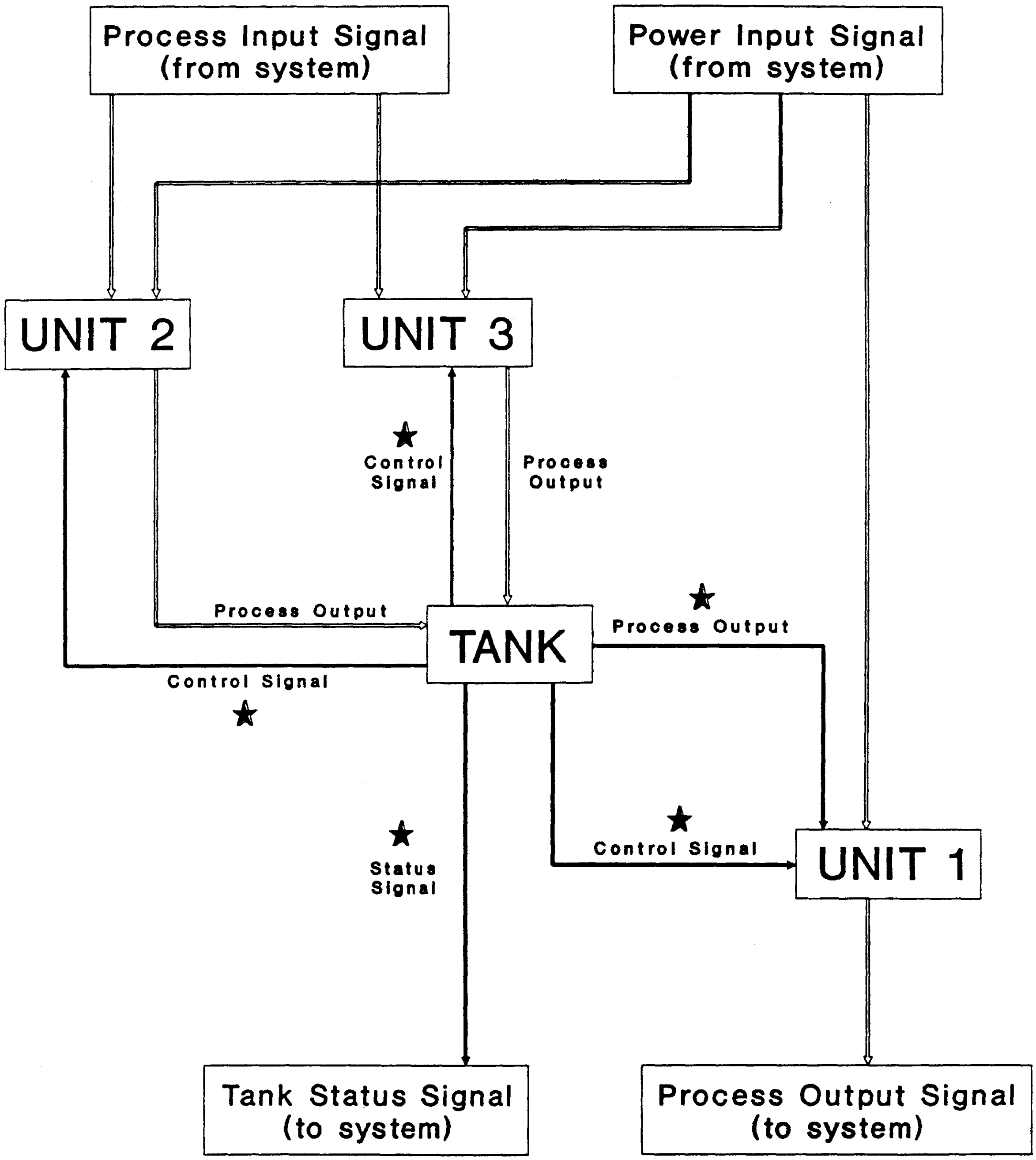


Figure 21. TANK PROGRAM SIGNALS

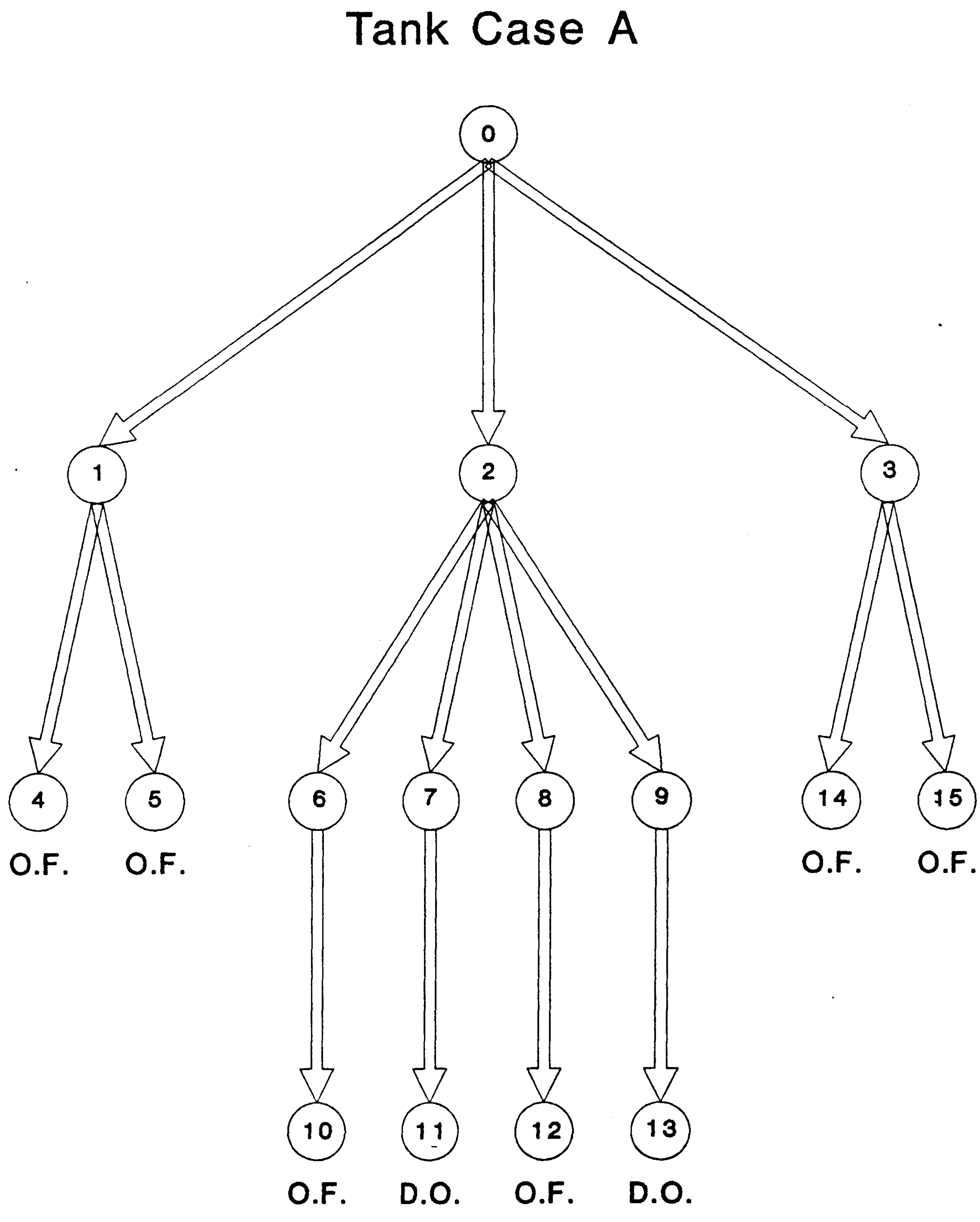


Figure 22. TANK CASE A STATE TRANSITION DIAGRAM

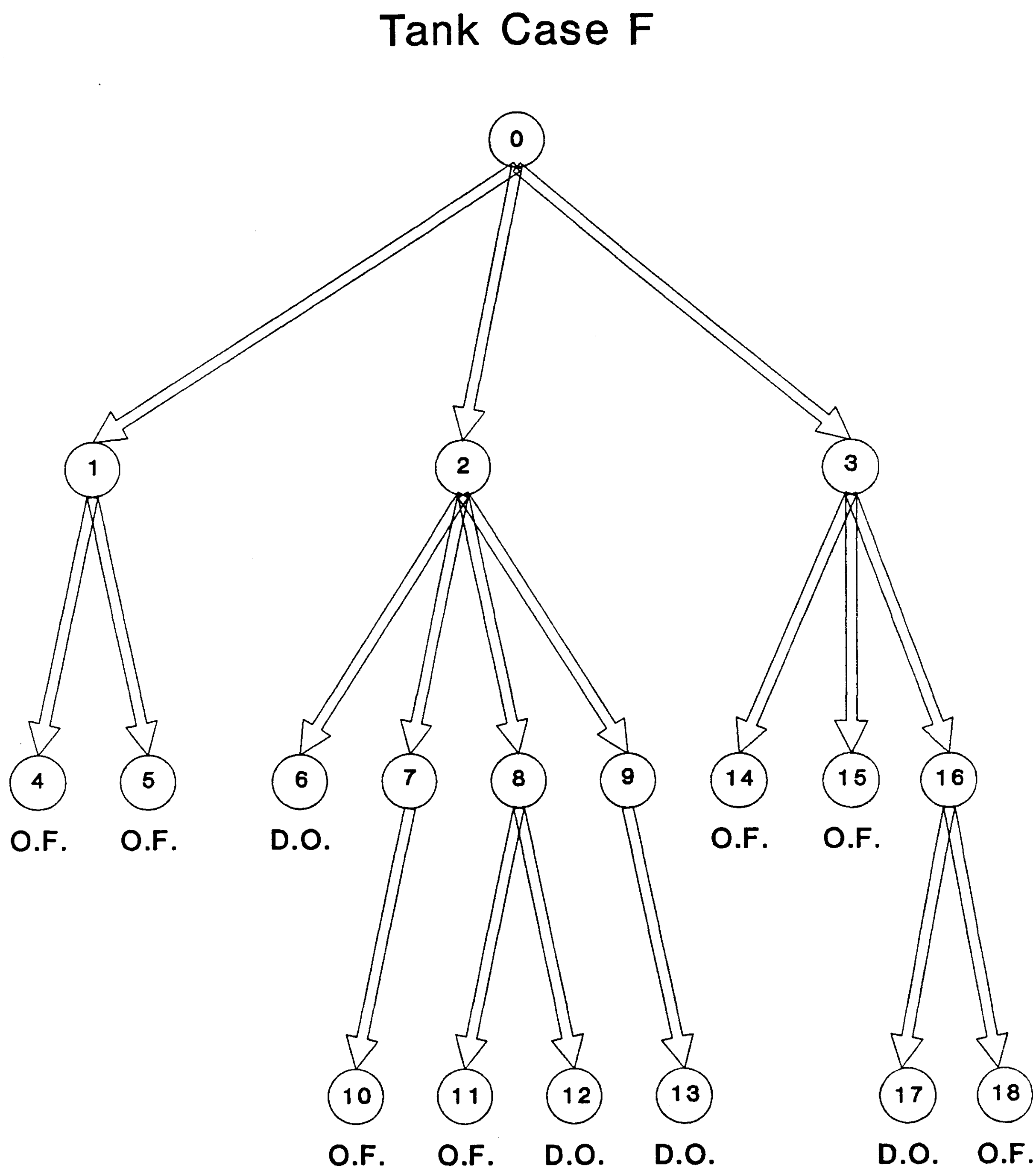


Figure 23. TANK CASE F STATE TRANSITION DIAGRAM

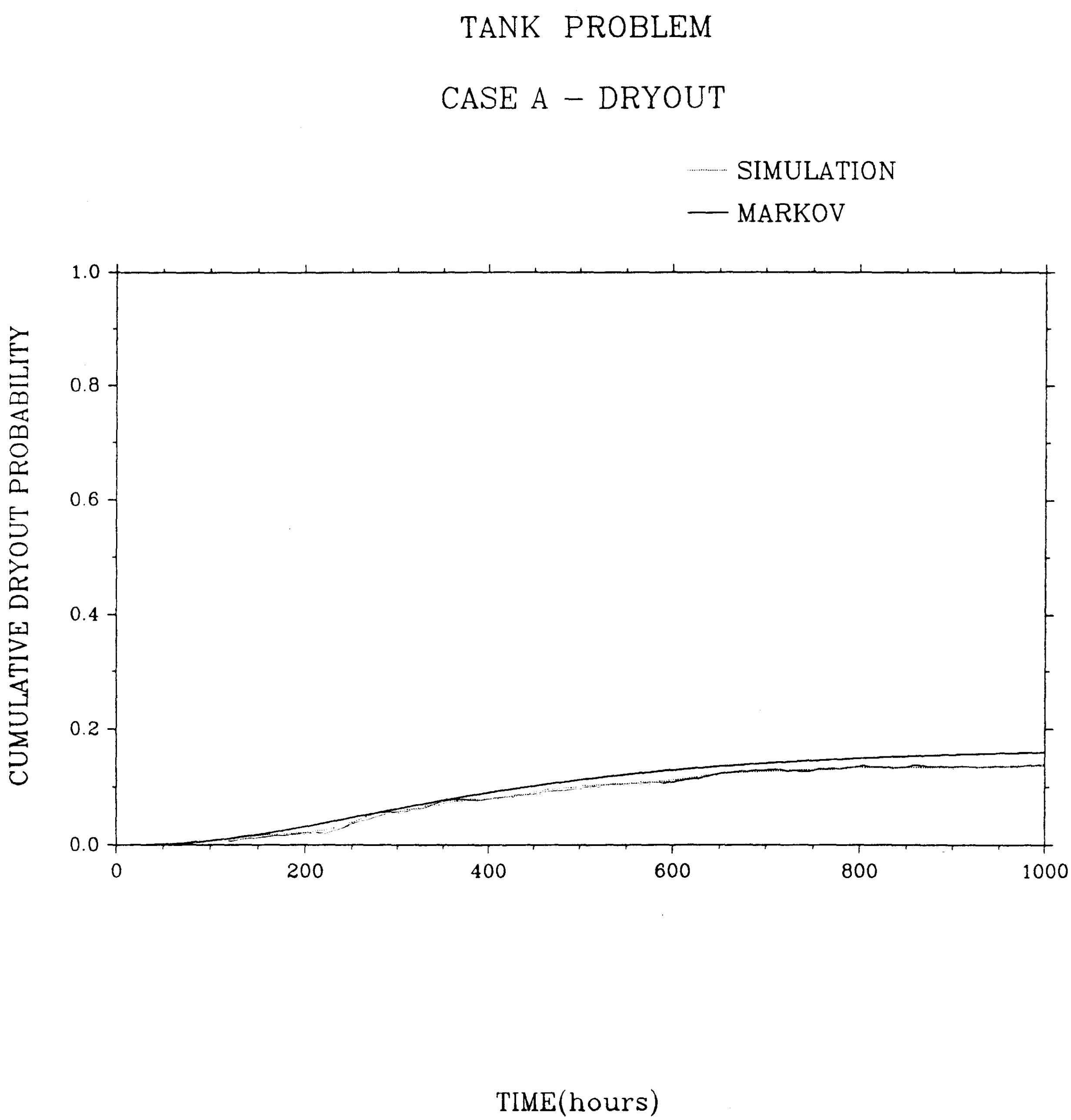


Figure 24. CASE A — CUMULATIVE DRYOUT PROBABILITY

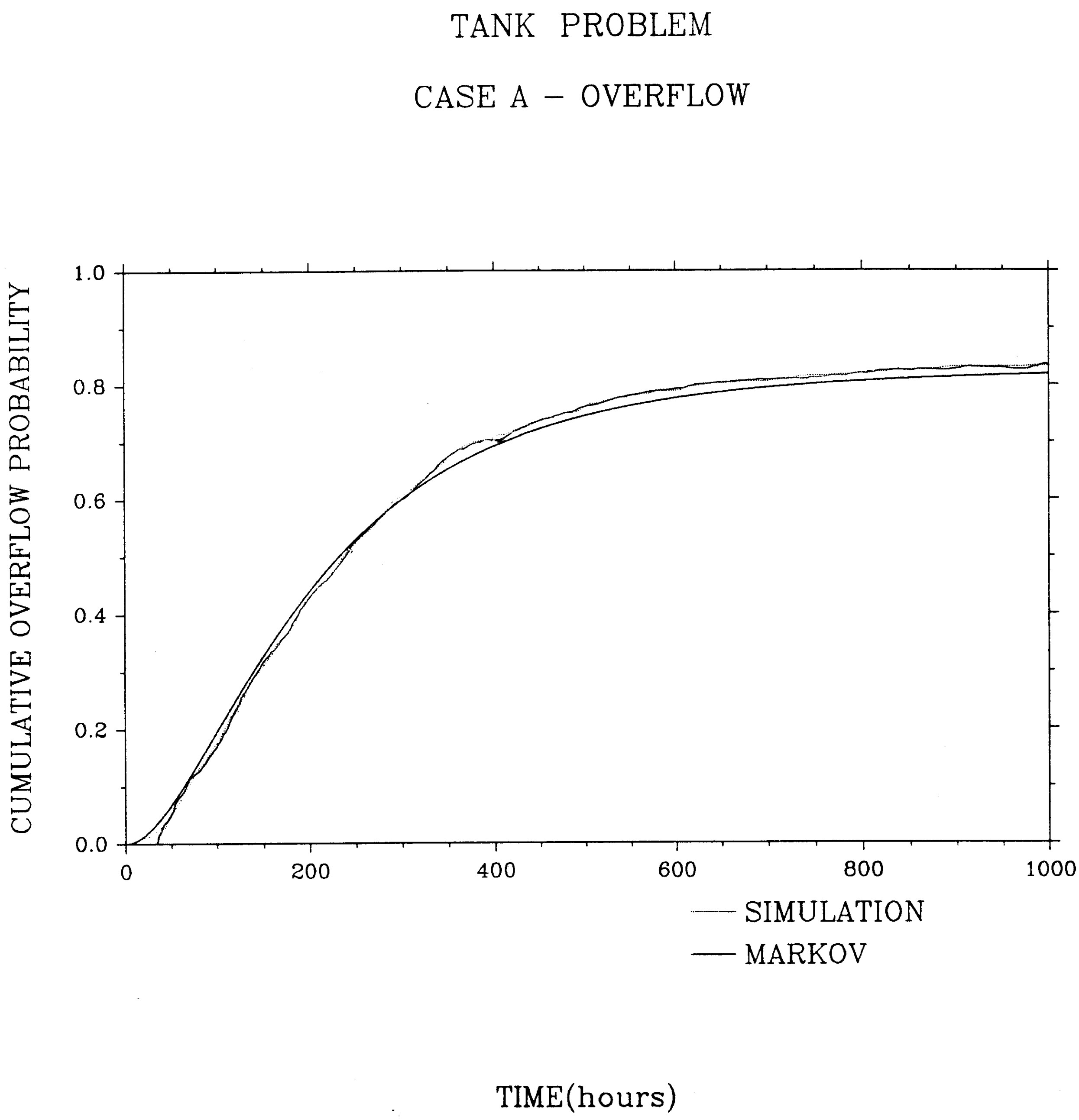


Figure 25. CASE A — CUMULATIVE OVERFLOW PROBABILITY

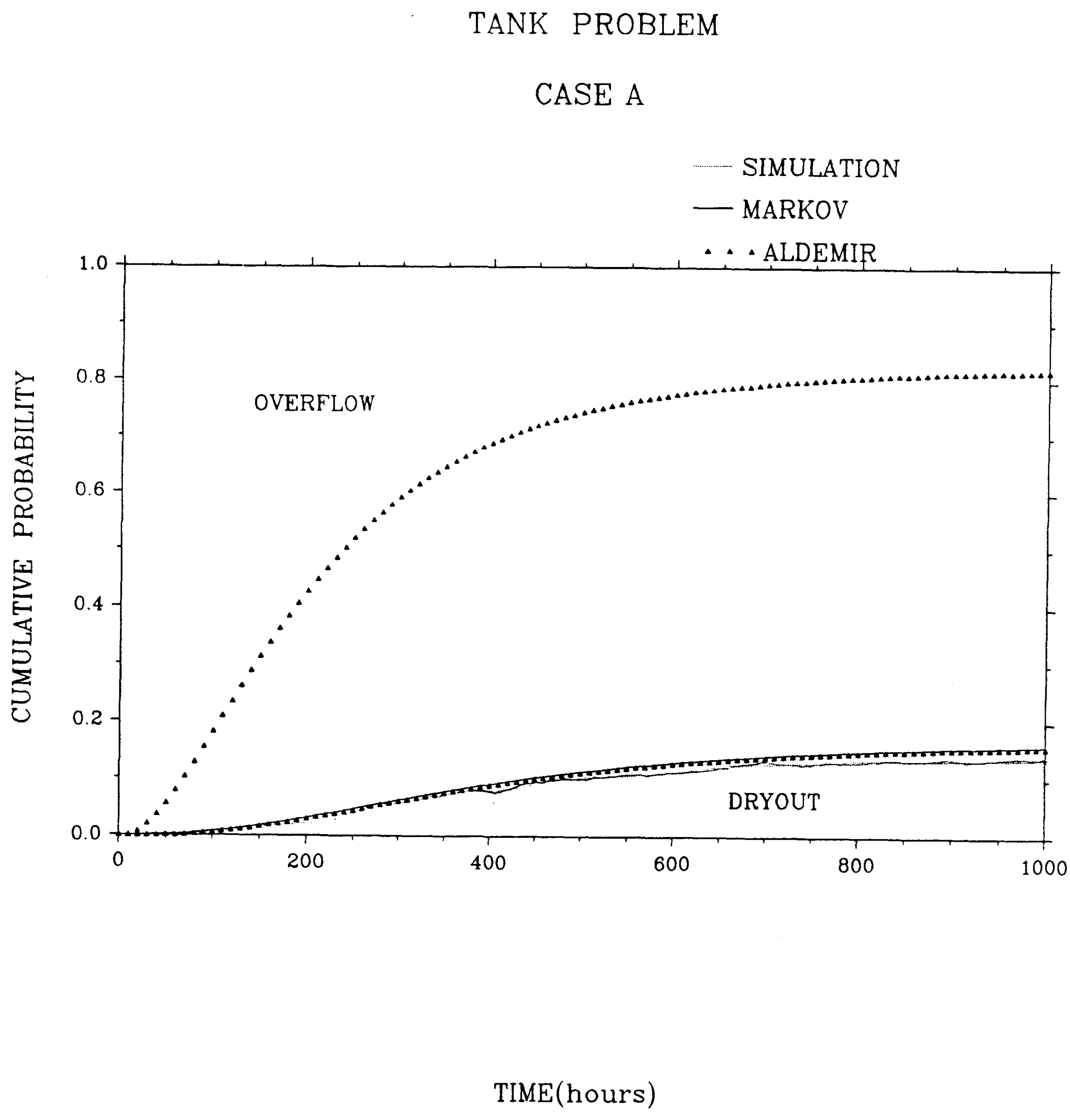


Figure 26. COMPARISON WITH REF. 10'S RESULTS FOR CASE A

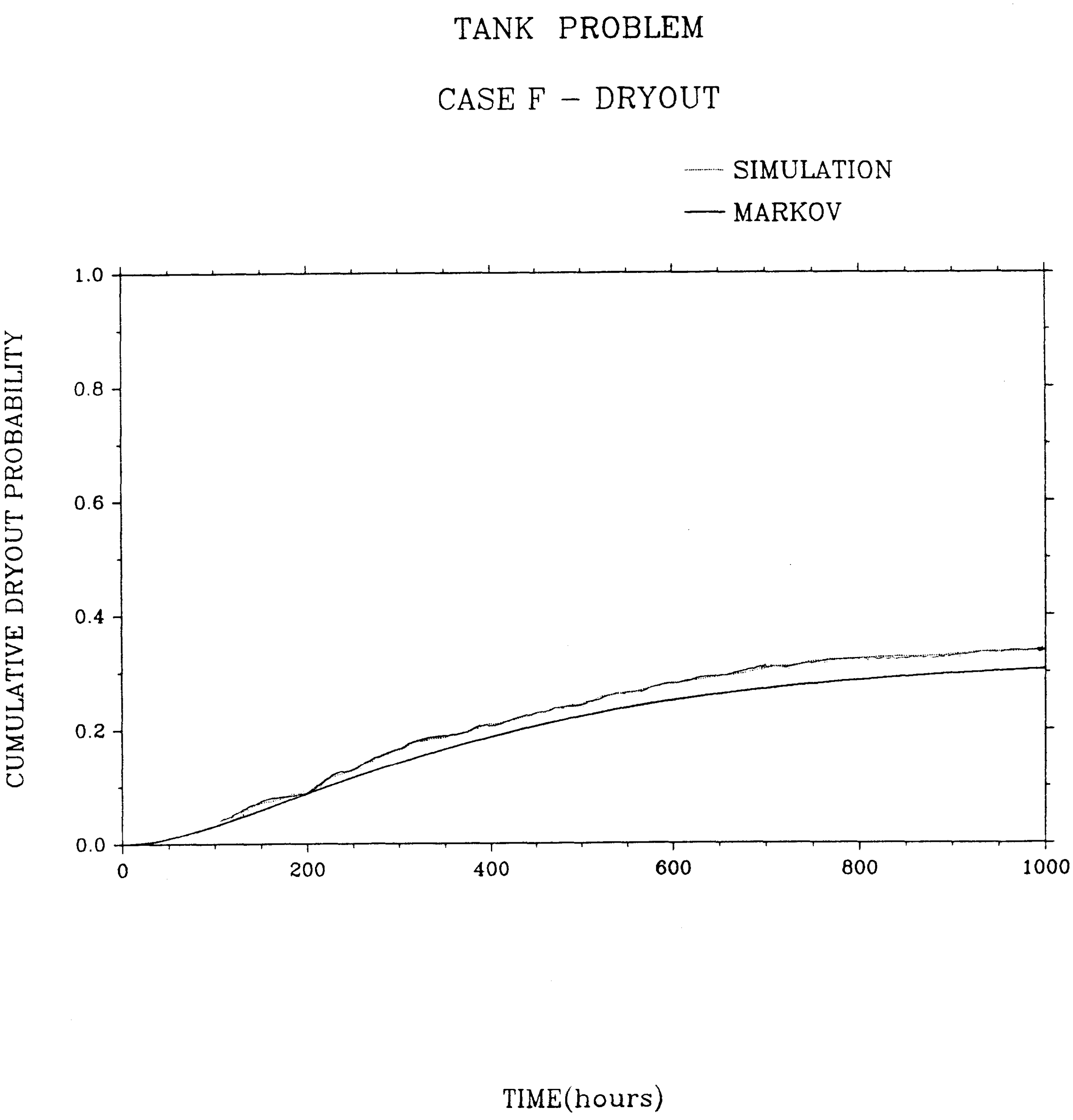


Figure 27. CUMULATIVE DRYOUT PROBABILITY

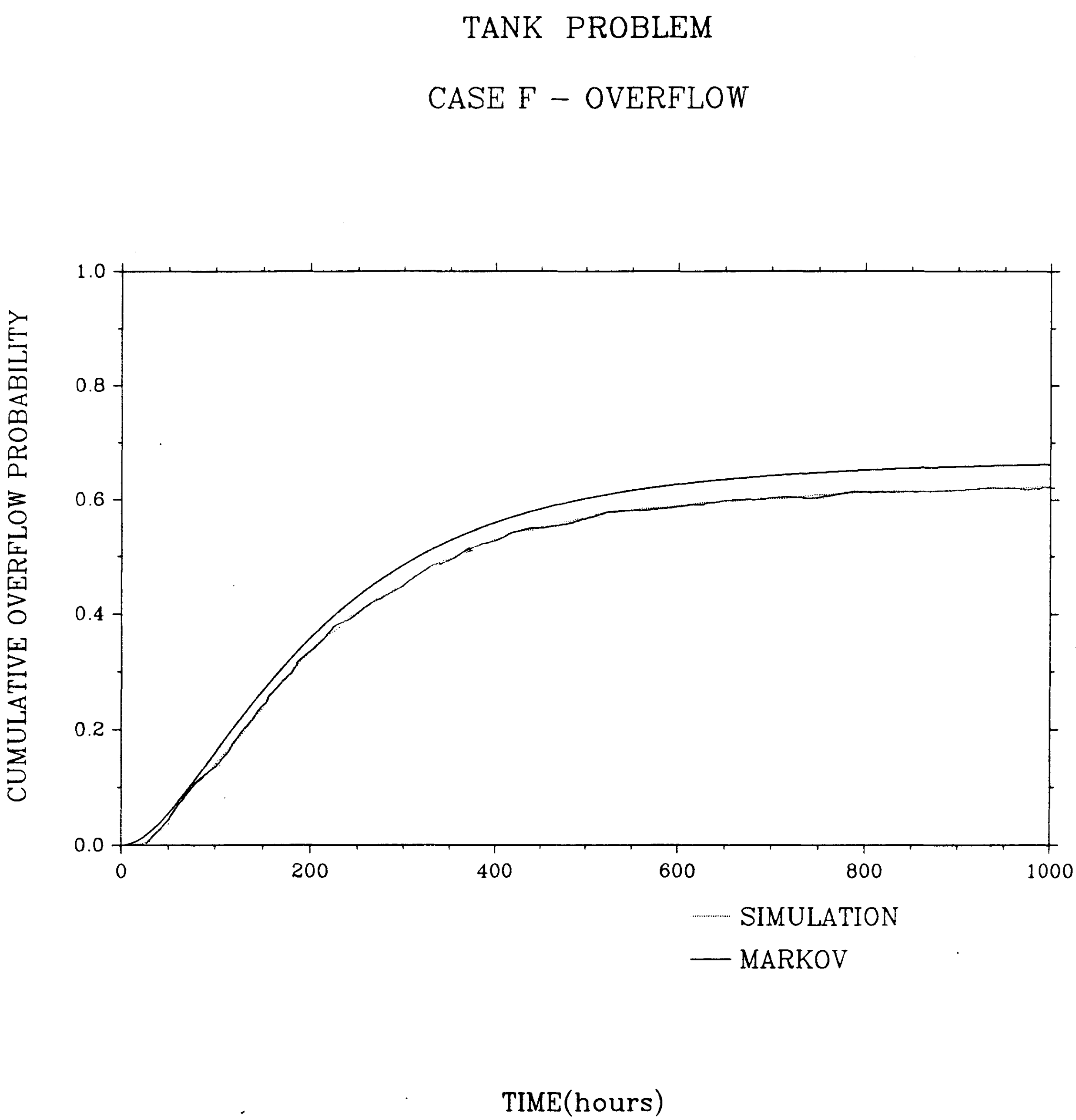


Figure 28. CUMULATIVE OVERFLOW PROBABILITY

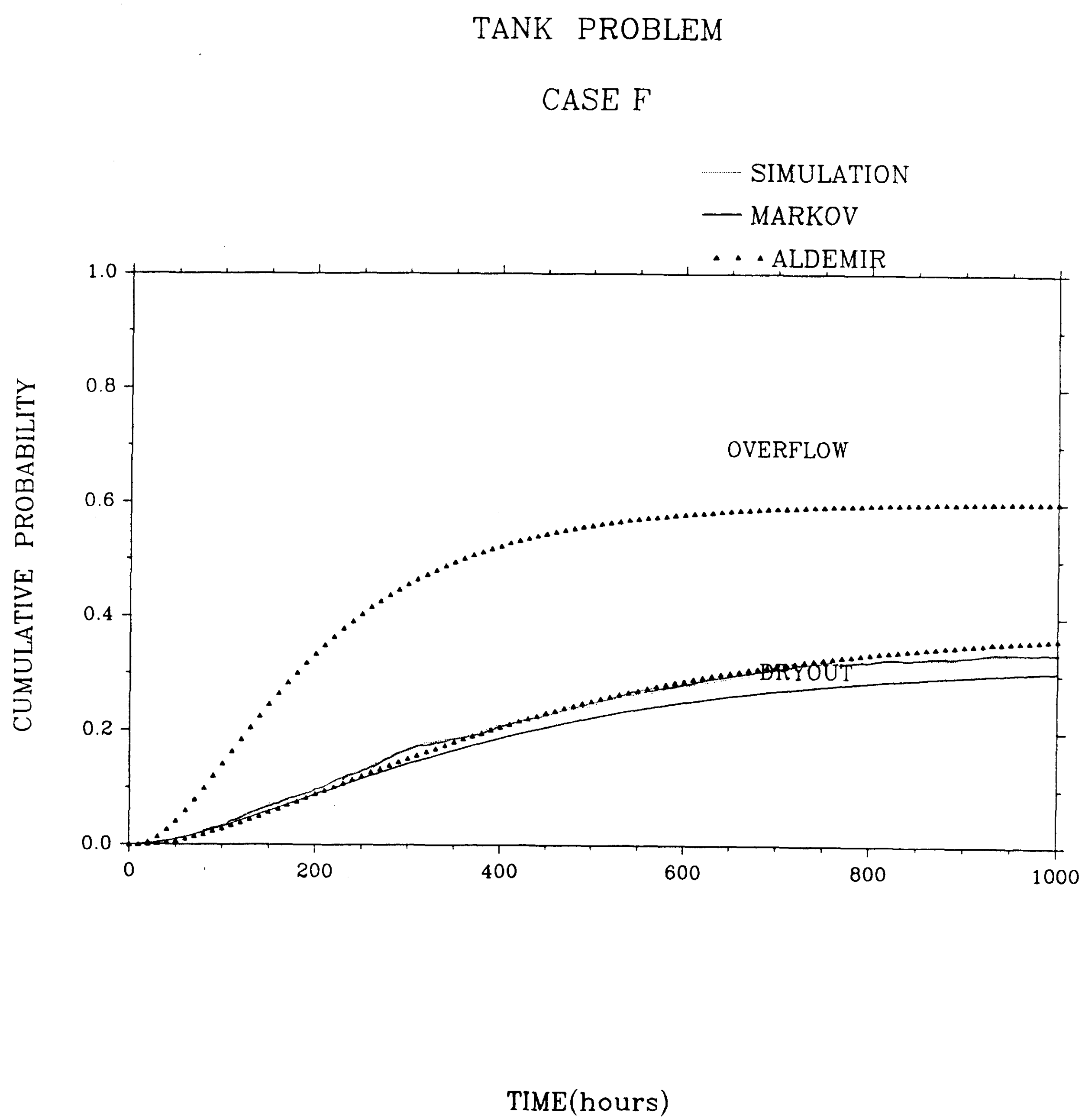


Figure 29. COMPARISON WITH REF. 10'S RESULTS FOR CASE F

5. SUMMARY AND CONCLUSIONS

In this work, a discrete event simulation program is developed for evaluating the dynamic availability of complex systems. The DYMCAM program is designed to be a general analysis tool with applicability to many types of engineering systems. The language used by the program, SIMSCRIPT II.5, provides event scheduling, process interaction, and continuous simulation capabilities, allowing: the treatment of components as separate objects within the program, the treatment of external events, and the analysis of process control systems. The latter capability is exploited in the TANK program, which is a small extension of DYMCAM.

The basic DYMCAM program is designed to allow the user to easily construct a system model in order to determine the system's time–dependent unavailability. The basic modeling entities are the components; the user specifies the components in the system, and the links between the components. Also specified are any external events (which can be used to perform a phased mission analysis). Five basic component types are presently available; however, further components can easily be added if called for. Program output includes instantaneous system unavailability at any number of user specified time points throughout the course of the simulated time period, and average unavailability information for the entire simulation.

The TANK code is a modified version of DYMCAM designed to demonstrate the capability for evaluating the unavailability of systems whose behavior is affected by continuous process variables. The TANK code provides the ability to model a continuously variable tank fluid level; it also demonstrates how a simulation program can be used to model the occurrence of events not scheduled before the start of the simulation.

Applications of the DYMCAM and TANK codes to various test problems indicate that the approach is reasonably accurate. They also demonstrate that the component–oriented approach adopted allows easy upgrading of the program to suit problem needs (e.g., non–exponential failure times, complex repair strategies, m–out–of–n logic). Even the incorporation of a continuous process variable into the base DYMCAM program does not require major restructuring of the base program (it does require the addition of a number of additional subroutines).

The major drawback of the discrete event simulation approach, as applied in the DYMCAM and TANK codes, is the large amount of computer time required to perform the analysis. This is due to a number of factors. First, the sampling done in these codes is of the "brute force" sort; a fair number of trials are required to obtain good accuracy. This problem will be greatly exaggerated when realistic failure rates are used. Second, the codes have been written to be understandable to the user, perhaps at the sacrifice of execution efficiency. Efforts to optimize the code are likely to lead to shorter running times, but also a model that is less easy to maintain and modify. Third, in order to determine the instantaneous unavailability during a simulation, the codes interrupt the simulation process. This reduces one of the advantages of discrete simulation – the bypassing of simulated time where no events are scheduled to occur. Reducing the number of time points at which the system unavailability will be estimated will speed up the simulation. Finally, the SIMSCRIPT II.5 implementation on personal computers is not really designed for heavy processing; this is because for these machines, the language is processed using an interpreter, rather than a compiler. Execution of the DYMCAM and TANK programs on minicomputers, or larger, should lead to much more acceptable run times.

Because of the runtime requirements of discrete event simulation (which will be significant – even on large computers – when dealing with realistic failure rates), future work in this area should concentrate on the issue of variance reduction, i.e., how to increase the accuracy of the unavailability estimates with a small number of samples. Once that is done, a number of refinements can be made to the programs, including incorporation of process signal strength between components (allowing a more natural treatment of m–out–of–n problems), expansion of the REPAIR.SUPERVISOR routine to accomodate various repair strategies, treatment of common cause failures, and incorporation of uncertainty analysis.

REFERENCES

1) A.E. Green and A.J. Bourne, *Reliability Technology*, Wiley, New York, 1972.

2) W.E. Vesely et al, Fault Tree Handbook, NUREG-0492, U.S. Nuclear Regulatory Commission, January 1981.

3) W. V. Gately and R. L. Williams, "GO Methodology – Overview," NP-765 Electric Power Research Institute, (1978).

4) A. Pages and M. Gondran, *System Reliability – Evaluation and Prediction in Engineering*, North Oxford Academic Publishers Ltd., 1986.

5) P.C. Cacciabue and A. Amendola, "Dynamic Logical Analytical Methodology Versus Fault Tree: The Case Study for the Auxiliary Feedwater System of a Nuclear Power Plant," *Nuclear Technology, 74*, 195–208(1986).

6) H. Kumamoto, T. Tanaka, and K. Inoue, "A New Monte Carlo Method for Evaluating System-Failure Probability," *IEEE Transactions on Reliability, R-36*, No. 1, (April 1987).

7) T. Matsuoka and M. Kobayashi, "GO-FLOW: A New Reliability Analysis Methodology," *Nuclear Science and Engineering, 98*, No. 1, (January 1988).

8) CACI, *SIMSCRIPT II.5 Programming Language*, CACI, Inc.-Federal, Los Angeles, 1987.

9) G. E. Apostolakis, S. L. Salem, and J. S. Wu, "CAT: A Computer Code for the Automated Construction of Fault Trees," NP-705 Electric Power Research Institute, (1978).

10) T. Aldemir, "Computer-Assisted Markov Failure Modeling of Process Control Systems," *IEEE Transactions on Reliability, R-36*, No. 1, (April 1987).

11) J. Banks and J. S. Carson II, *Discrete-Event System Simulation*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

12) J. Banks and J. S. Carson II, "Process-interaction Simulation Languages," *Simulation, 44*, No. 5, (May 1985).

13) D.L. Deoss, "A Simulation Model for Dynamic System Availability Analysis," S.M. Thesis, Massachusetts Institute of Technology, 1989.

14) T.J. McIntyre and N. Siu, "Electric Power Recovery at TMI-1: A Simulation Model," Proceedings of the International ANS/ENS Topical Meeting on Thermal Reactor Safety, San Diego, CA, February 2-6, 1986, pp. VIII.6-1 through VIII.6-7.

15) A. M. Law and C. S. Larmey, *An Introduction to Simulation Using SIMSCRIPT II.5*, CACI, Inc.-Federal, Los Angeles, 1984.

16) E. C. Russell, *Building Simulation Models with SIMSCRIPT II.5*, CACI, Inc.–Federal, Los Angeles, 1983.

17) CACI, *PC SIMSCRIPT II.5 Introduction and User's Manual*, Third Edition, CACI, Inc.–Federal, Los Angeles, 1987.

18) A. M. Fayek, *Introduction to Combined Discrete–Continuous Simulation Using PC SIMSCRIPT II.5*, CACI, Inc.–Federal, Los Angeles, 1988.

19) W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes*, Cambridge University Press, New York, 1986.

20) T. Aldemir, "Quantifying Setpoint Drift Effects in the Failure Analysis of Process Control Systems," *Reliability Engineering and System Safety*, *24*, No. 1, (1989).

21) T. Aldemir, personal communication to N. Siu, Massachusetts Institute of Technology, April 10, 1989.

Appendix A
DYMCAM INPUT FILE DESCRIPTION

Figure A.1 shows an example listing of an input file for the DYMCAM program. Line numbers are indicated to aid in describing the setup of an input file for a specific problem, since different problems will require different numbers of input data file lines. Any text editor can be used to create the input file, and the file can be given any name acceptable by DOS requirements.

Line 1 is the title line and can be up to 80 characters long. If the title is less than 80 characters long, it will be necessary to enter spaces to extend the line to the full length. The read statements in the Input routine are formatted reads, and therefore, if 80 characters are not found on the first line, the program will look on the next line for the remaining characters of the title, thus misreading the desired input data contained in later lines. Some text editors, such as K–EDIT, do not save the trailing blank spaces and thus could cause a problem if attempts are made to use them to create input files. One trick that can be used if the title is short, is simply to enter spaces out to column 80 of line 1, and then enter a character in column 81. K–EDIT will save the entire line, but DYMCAM will only read the first 80 characters of the line, thus printing only the title desired.

Line 2 contains the number of simulated hours for which the program is to be run. The format is d(10,2) which means the program is looking for a decimal number with two digits after the decimal point, and that the number will be found in the first ten columns of line 2. For this particular format specification it is not necessary to have the value right justified in the ten column field of interest. The value can be entered left justified, if desired, and the program will read all digits to the left of the decimal point as an integer value and then read the next two digits following the decimal and ignore any other characters which may be in the first ten columns. It is critical only that the decimal appear somewhere in the first 10 columns so that format specifications are satisfied. If the decimal appears in columns 9 or 10, then the one or two digits following the decimal for which values have not been assigned will be recorded as being zero. This is true for any number read with a d(x,y) format. Regardless of the value of y, as long as a decimal is somewhere in the x columns specified, then y characters will be assigned following the decimal. If y characters are present in the input field, then they will be entered, if not, then zeroes will be entered for the remaining digits. For the example shown, the input value of simulation time is 1000 hours.

Line 3 is an integer value and must be entered in column 10. The value which may be entered is either a 0 or a 1. The 0 entry signifies that the run is to be a normal run. The 1 entry indicates that the run will be a test run to see if proper program operation is occurring. Entering a 1 will cause all components to fail at their mean failure time (one over the failure rate) and all repairs to occur at their mean repair time. Thus by entering a 1, it is possible to check and make sure that all components are failing and being repaired as expected. The example shown in Figure A.1 has a 0 entered indicating the run will be a normal run.

Line 4 indicates the number of Monte Carlo trials to be performed. The number is entered as an integer value and must be right justified so that the right most character of the number is entered in column 10 of line 4. The example shows a value of 1000.

Line 5 specifies the number of time points for which dynamic system unavailability data is required. This number is also an integer value and must also be right justified with the one's digit falling in column 10. There is no requirement as to the number of time points to be entered. If desired, a zero can be entered and no dynamic information will be calculated for the system. For the example problem, 11 time points will be used for the dynamic unavailability analysis.

Line 6 is an integer value referring to the manner of specifying the time points at which the instantaneous system unavailability will be estimated. The integer numbers 0, 1, or 2 must be entered in column 10 of line 6. Entering a 0 indicates that the next lines of the input file will contain the desired time points. For the example of Figure A.1, a value of 0 is specified, indicating that the next 11 (number of time points specified in line 5) lines of the input line will contain the time points of interest. If a 1 had been entered, then the 11 time points would have been chosen as uniformly distributed between time zero and the value specified in line 2. If a value of 2 is entered in column 10 of line 6, then the program will choose values for the time points which are logarithmically distributed between the zero time and the end of simulation time specified in line 2. This feature may be useful for evaluating the unavailability of a system which is suspected of having an exponentially distributed result. Since the time required to run the simulation program is directly related to the number of time the program is interrupted to take another time dependent unavailability sample, it is desireable to keep the number of time points specified in the input file to a minimum, while still providing sufficient data to properly evaluate the dynamic behavior of the system.

Line 18 of the input file specifies the number of components contained in the system. For the example the number of components is 2. This value will always be an integer value and must be entered right justified with the right most digit falling in column 10. For every component indicated by this number, there will be a minimum of five line of data in the input file. For the example of Figure A.1, the first component is described in lines 19 through 23 and the second component is described in lines 24 through 30.

Each component must have a first line entered in the format of lines 19 and 24. The first 10 columns are reserved for the components name. The name can contain any characters desired, but must not contain spaces. It need not be left or right justified. It need only be less than or equal to 10 characters in length. The SIMSCRIPT language distinguishes between small and capital letters; therefore it is important that if capital letters are used for component names, that this is done consistently every where a specific component name is mentioned. All other text, other than component names, must be entered in lower case letters, since this is what the DYMCAM program has been programmed to recognize.

Columns 11 through 20 for the first line of each component must contain the component type designation. This, as all text, need not be justified, but must be in lower case letters. Columns 21 through 30 should contain the component's initial state upon execution of the simulation. This information must also be in lower case letters. Also on this line, the number of input and output signals used by the component should be specified. Any number of input and output signals can be assigned to a given component. However, for all components, at least one input and one output signal must exist. The number of inputs is an integer value and must be right justified in the column 31 to 35 field, while the number of output signals must be entered as an integer value right justified in the 36 to 40 column field. Line 19 of the example refers to a passive element named BATTERY which is initially in standby at time zero and has one input and one output signal. Line 24 indicates a switch named SWITCH which is initially open and has three inputs and one output signal.

The second line of each component data field (lines 20 and 25 of the example) contains the failure data for the component. The first 10 columns contain the demand failure probability. The format for reading this value is d(10,5). The second data field of this line is from column 11 to column 20. This will contain the failure rate (λ) for the component. The format for this value is also d(10,5).

The third line for each component (lines 21 and 26) must contain the repair information. Three data values are entered and each is read in the d(10,5) format. The first value is the α–parameter for the Weibull distribution and it must be found in columns 1 through 10. The second value is the β–parameter and must be entered in columns 11 through 20. The third value is the probability the component is repairable once it has failed. This number is entered in columns 21 through 30. If exponentially distributed repair is to be considered, this can be accomplished by entering a 0 for the value of α and using a β equal to the mean repair time for the component (one over μ, the exponential repair rate). For cases when a 1 is entered in line 3 of the input file, the mean time to repair is treated as being equal to the Weibull parameter, β, regardless of the value of the α. For the example shown, repair is not considered, thus the values entered in lines 21 and 26 do not have physical significance, except for the zeros, which simply indicate that once the component fails, it stays failed since it is not repairable.

For every signal in the system, a line like lines 22, 23, and 27 through 30 must be specified. Since signals must be associated with the components they link, they will always be listed following the component. The number of signals described following any component will equal the sum of the number of input and output signals specified for the given component. For the example shown, the BATTERY has one input and one output, thus two signals are specified. For the SWITCH, there are three inputs and one output, thus four signals are specified. The input signals for a component must always be specified first and the output components last. The order of specifying several input or output files for a given component, however, is not important as long as the above rule is obeyed. Every signal which does not originate from, or terminate at the system level, must be contained in two component listings, since each signal must have an origin and a destination.

Information concerning signals must always begin in the column 11 to 20 field. The first 10 columns are blank for ease in viewing. The first field (columns 11 to 20) attaches the signal to another component. For input signals, this field contains the name of where the signal came from (either the system or an other component), and for output signals the data field contains the destination of the signal (either the system or an other component).

The second data field for each component is contained in columns 21 through 30 and indicates the type of signal (either command, power, or process). The third piece of data concerning each signal is its strength at the start of the simulation. For power and process signals the strength is 0 if power is not available or the process variable is not present, and the strength is 1 if power is available or the process variable exists. For command signals, a value of 0 indicates no command, while a value of 1 indicates a signal to open the switch or valve (or start the active component). A value of –1 indicates a signal to close the valve or switch (or to stop the active component). These values are entered as integers and are right justified in column 35.

For the example of Figure A.1, the BATTERY has one input and one output signal. The input is a process signal coming from the system and is initially off, while the output signal is a process signal going to the SWITCH and is also initially off. The SWITCH has three inputs and one output. Two of the inputs are from the system and reflect the power and command signals to the SWITCH. Initially the switch has power but no command signal. The other input to the switch is the process signal which comes from the BATTERY. The output signal is a process signal which goes to the system.

Line 31 provides information about the initial state of the system. The program does not calculate the system state until time 0+, which is slightly greater than time zero. Thus, to artificially set the system to its desired initial operating state, it is necessary to set it at the beginning of the run. For the system to be available at time zero, the system status is set to operating or standby. Thus the value entered for initial system state is either operating, standby, or failed. This data is entered in the first 10 columns of the input file line. Line 31 of the example indicates the system initially starts in the standby condition.

The next required line in the input file is the system success criteria. This is the number of output signals directed to the system which must be on in order for the system to be considered available. It is entered as an integer value and must be right justified in column 10 of the data line. For the example, the value entered in line 32 is one, specifying that at least one output signal to the system must be on in order for the system to be available. For this example, there is only one output signal to the system (the output process signal from the switch), and so the system is only available if the switch is closed and an output process signal is being generated, i.e. the BATTERY must also be operating.

Next, the number of external events to be included in the problem scenario must be entered. This value will be an integer and is read right justified from column 10 of the data file line. This value may be zero if the problem to be analyzed is not a phased mission one; if this is the case, this will be the last line of the input file. For the example of Figure A.1, line 33 indicates that there are 3 external events for this problem.

For each external event, at least four lines of data must be entered. The first line contains the time at which the event is scheduled to occur. This information is contained in columns 1 to 10 and is read in the d(10,2) format. Following this, in columns 11 to 20, the number of components affected by the external event are given. This is an integer value and must be entered right justified in column 20. Every external event must affect at least one component or signal, but not necessarily both, therefore this value may often be 0 as it is in lines 34 and 38 of the example. If the value is 1 or greater, then the next lines will list the components effected by the external event. Each line, like line 43 of the example, simply lists the name of the affected component. The name must be found in the first 10 columns of the data file line. For the example, the external event changes the state of the SWITCH. The program is written such that all components changed by a given external event, are affected in the same manner. Thus the next data file line following the component names gives the new state of these components. For the example, the external event opens the SWITCH at 900.00 hours into the simulation. Thus line 44 contains the instruction to open. This component change of state must be entered in the first 10 columns of the data line.

The next line of an external event specifies the number of signals affected by the event. This will be an integer value and must be entered right justified in column 10 of the data line. For the example of Figure A.1, the third external event does not change any signals as is indicated by the 0 in line 45. The first two external events change one signal each. This is indicated in lines 35 and 39 of the example input file. If a signal is changed, then two lines must be entered for each signal changed by the external event. The first line contains the origin of the signal, the destination of the signal, and the type of signal. These three data entries are text information and are entered in columns 1 to 10, 11 to 20, and 21 to 30 respectively. The next input data line contains the new strength of the signal. This will be an integer value and is entered right justified in column 10 of the data file line. For the example of Figure A.1, the first external event changes the process signal from the system to the BATTERY (line 36). The new strength (line 37) specifies that the signal is to be turned on so that the BATTERY may now supply current. The second external event of the example affects the command signal from the system to the SWITCH. It causes the command signal to change to –1 at the 500.00 hour time point which will cause the switch to close (provided it does not experience a demand failure). Line 40 of the example specifies the signal, while line 41 gives the new value.

With the current program structure, it is possible to change many signals with a single external event, and to change each to a different signal strength. These same signals may be changed again at a later time in the simulation by another external event. Components, on the other hand, can only be changed once by an external event. This means that if an external event is used at the 500.00 hour time point to open a switch, the same switch can not be closed with an external event at a later time in the simulation (although it may have its input command signal changed). This is because of the way external events were treated in development of this basic demonstration program. It would be possible to modify the program to allow multiple state changes of a given component, if such a capability were desireable.

Also with the current structure, all components changed by a given external event must be changed to the same new state. This is not such a problem since any number of external events can be scheduled to occur at exactly the same time. In fact, the motivating idea for the external event was that each event would effect only a single component or type of component. If it is desireable, the EXTERNAL.EVENT routine could certainly be modified to allow multiple component changes during a single external event.

This appendix should supply all the information necessary for writing input files for the DYMCAM program. Care must be taken to ensure that all information is properly formatted. For further examples of input files, Appendix D can be consulted which contains several input files used for the various test runs performed in Sections 3 and 4. Also note in Appendix D that all data file lines (with the exception of the title line) contain data only up through column 40. Since SIMSCRIPT will not look beyond this point for any data, it is possible to use this "blank space" to include comments concerning the input file data for future reference and ease of understanding. This has been done for all test cases run.

LINE NUMBER	INFORMATION							
1	Test Simulation Program							
2	1000.00							
3	0.00							
4	1000.00							
5	11.00							
6	0.00							
7	0.00							
8	100.00							
9	200.00							
10	300.00							
11	400.00							
12	500.00							
13	600.00							
14	700.00							
15	800.00							
16	900.00							
17	1000.00							
18	2.00							
19	BATTERY		passive		operating		1	1
20		0.1		0.0				
21		1.0		1.0		0.0		
22			system		process		0	
23			SWITCH		process	1		
24	SWITCH		switch		open		3	1
25		0.3		0.0				
26		1.0		1.0		0.0		
27			system		command		0	
28			system		power		1	
29			BATTERY		process		0	
30			system		process		0	
31	standby							
32		1.0						
33		3.0						
34		100.00		0.0				
35		1.0						
36	system		BATTERY		process			
37		1.0						
38		500.00		0.0				
39		1.0						
40	system		SWITCH		command			
41			–1.0					
42		900.00		1.0				
43	SWITCH							
44	open							
45		0.00						

Figure A.1 – Example DYMCAM Input File

Appendix B

DYMCAM Program Listing

```
preamble
''
''   RISK - Test program to simulate system behavior
''
''   03/28/89
''
   permanent entities

      every component.record
         has a component_name,
            a component_type,
            a number_inputs,
            a number_outputs,
            a response_function,
            an initial_state,
            a demand_failure_frequency,
            a run_failure_frequency,
            a repair_probability,
            a repair_function_shape, and
            a repair_function_scale

      every external.event.record
         has an occurrence_time,
            a number_components,
            a new_state,
            a number_signals, and
            a new_strength

      define response_function as a subprogram variable
      define component_name, component_type, initial_state,
         and new_state as text variables
      define demand_failure_frequency, run_failure_frequency,
         repair_probability, repair_function_shape,
         and repair_function_scale as real variables
      define number_inputs, number_outputs, number_components,
         number_signals, and new_strength as integer variables
''
''   2-d arrays associated with permanent entities.
''
   define input.name, output.name, input.signal.type,
      output.signal.type, extevnt.component, extevnt.origin,
      extevnt.destination, and extevnt.stype
      as 2-dimensional text arrays
   define input.signal.strength and output.signal.strength
      as 2-dimensional integer arrays
   define test as a 1-dimensional text array
   define signal.status as a 1-dimensional integer array

   processes include call.update, schedule.avail.samples,
      schedule.external.events, repair.supervisor,
      stop.tank, and stop.scenario                            ''TANK

      every component
         has a name,
            a component.type,
```

```
            a response.function,
            an old.state,
            a state,
            a demand.failure.frequency,
            a run.failure.frequency,
            a repair.probability,
            a repair.function.shape,
            a repair.function.scale,
            a failure.time,
            a status,
         and owns an input.sset and
            an output.sset
         and may belong to a system.cset,
            a tank.input.cset,                                     ''TANK
            a tank.output.cset,                                    ''TANK
            and an extevnt.cset

      every tank                                                   ''TANK
         has a high.level,                                         ''TANK
            a low.level,                                           ''TANK
            a high.set,                                            ''TANK
            a low.set,                                             ''TANK
            a level,                                               ''TANK
            a flow.rate.in,                                        ''TANK
            a flow.rate.out,                                       ''TANK
            a net.flow.rate,                                       ''TANK
         and owns a tank.input.cset,                               ''TANK
            a tank.output.cset,                                    ''TANK
            a tank.input.sset, and                                 ''TANK
            a tank.output.sset                                     ''TANK
         and belongs to a system.tset                              ''TANK

      every external.event
         has an occurrence.time,
            a new.state,
            a number.signals,
            a signal.origin,
            a signal.destination,
            a signal.typee, and
            a new.strength
         and owns an extevnt.cset
         and belongs to a system.eset

      every availability
         has a time.avail, and
            a time.avail.data

   define time_avail as a 1-dimensional real array
   define time.avail and time.avail.data as real variables
   define tank.condition as an integer function                    ''TANK
   define response.function as a subprogram variable
   define name, component.type, old.state, state, new.state,
      signal.origin, signal.destination, and signal.typee
      as text variables
   define demand.failure.frequency, run.failure.frequency,
```

```
      repair.probability, repair.function.shape,
      repair.function.scale, failure.time, occurrence.time,
      high.level, low.level, high.set, low.set,                   ''TANK
      flow.rate.in, flow.rate.out, net.flow.rate,                 ''TANK
      and number.signals as real variables
   define status and new.strength as integer variables
   define level as a continuous double variable                   ''TANK
''
''  Later versions may define signals as processes (so time delays
''  can be built in).
''
   temporary entities

      every signal
         has a signal.type,
            an origin,
            a destination,
            an old.strength, and
            a strength
         and may belong to an output.sset,
            an input.sset,
            a tank.input.sset,                                    ''TANK
            a tank.output.sset,                                   ''TANK
            a system.boundary.sset,
            a system.success.sset, and
            a system.sset

   define cptr, sptr, eptr, aptr, and tptr                        ''TANK
      as 1-dimensional pointer arrays

   define signal.type, origin, and destination as text variables
   define old.strength and strength as integer variables
''
''  System characteristics.
''
   the system owns a system.boundary.sset,
      a system.success.sset,
      a system.cset,
      a system.sset,
      a system.eset, and
      a system.tset                                               ''TANK

   define failure.translation as a text function
   define job.title, initial.system.state, and system.state
      as text variables
   define system.ind.var and simulation.time as real variables
   define ntrial, system.success.criterion, ntimes,
      distribution.type, run.type, and total.signal.count
      as integer variables
   define unavailability.dist as a 1-dimensional real array
   define trial.unavail as a real variable

   accumulate trial.availability as the mean of system.ind.var
   tally average.unavailability as the mean,
         variance.unavailability as the variance,
```

```
            maximum.unavailability as the maximum,
            and minimum.unavailability as the minimum of
            trial.unavail

      define .off to mean 0
      define .on to mean 1
      define .no to mean 0
      define .yes to mean 1
      define .working to mean 1
      define .resetting to mean 2
      define .awaiting.repair to mean 3
      define .under.repair to mean 4
      define .not.repairable to mean 5
      define .reset.run to mean 6

end ''preamble
```

```
main
   define trial as an integer variable
''
''   Problem input
''
   call input
   call run.initialize
   call tank.initialize.run                                       ''TANK
   add .003 to simulation.time
   for trial = 1 to ntrial
   do
      call trial.initialize
      call tank.initialize.trial                                  ''TANK
      activate a call.update now
      activate a schedule.avail.samples now
      activate a schedule.external.events now
      activate a stop.tank in simulation.time hours               ''TANK
      activate a stop.scenario in simulation.time hours
      start simulation
      let unavailability.dist(trial) = 1 - trial.availability
      let trial.unavail = trial.availability                      ''TANK
      let time.v = 0
      reset totals of system.ind.var
   loop

   call run.output

end ''main
```

```
routine active given component
''
''   Develops output signals for an active component
''   using explicit command signals.  Assumes that the component
''   has one or more command signal inputs, power inputs, and
''   process inputs:
''
''        input command  ---|‾‾‾‾‾‾‾‾‾‾‾|
''        input power    ---|           |--- output process
''        input process  ---|___________|
''
''   Condensed decision table:
''
''              Command   Power  Process  Initial     Final      Process
''      Case    Input     Input  Input    State       State      Output
''      ----   -------   -----  -------   -----       -----      ------
''       1       -         -       -      failed      failed      no
''       2       -        no       -      standby     standby     no
''       3     stop       yes      -      standby     standby     no
''       4     none       yes      -      standby     standby     no
''       5     start      yes     no      standby     standby*    no
''                                                    failed      no
''       6     start      yes     yes     standby     standby*    no
''                                                   operating    yes
''       7       -        no       -     operating    standby     no
''       8     stop       yes     no     operating    failed      no
''                                                    standby     no
''       9     stop       yes     yes    operating   operating*   yes
''                                                    standby     no
''      10     none       yes     no     operating    failed      no
''      11     none       yes     yes    operating   operating    yes
''      12     start      yes     no     operating    failed      no
''      13     start      yes     yes    operating   operating    yes
''      14       -         -       -      standby*    standby*    no
''      15       -        no       -     operating*  operating*   no
''      16       -        yes     no     operating*   failed      no
''      17       -        yes     yes    operating*  operating*   yes
''
   define rule as a saved 2-dimensional text array
   define component as a pointer variable
   define index.command, total.command, number.power, total.power,
      number.process, total.process, output.strength, ruletype,
      success, and j as integer variables
   define later.case as a saved integer variable
''
''   Enter decision table.
''
   if later.case eq .no
      reserve rule as 17 by 4
      let rule(1,1) = ""       let rule(1,2) = ""
      let rule(1,3) = ""       let rule(1,4) = "failed"
      let rule(2,1) = ""       let rule(2,2) = "no"
      let rule(2,3) = ""       let rule(2,4) = "standby"
      let rule(3,1) = "stop"   let rule(3,2) = "yes"
      let rule(3,3) = ""       let rule(3,4) = "standby"
```

```
      let rule(4,1) = "none"   let rule(4,2) = "yes"
      let rule(4,3) = ""       let rule(4,4) = "standby"
      let rule(5,1) = "start"  let rule(5,2) = "yes"
      let rule(5,3) = "no"     let rule(5,4) = "standby"
      let rule(6,1) = "start"  let rule(6,2) = "yes"
      let rule(6,3) = "yes"    let rule(6,4) = "standby"
      let rule(7,1) = ""       let rule(7,2) = "no"
      let rule(7,3) = ""       let rule(7,4) = "operating"
      let rule(8,1) = "stop"   let rule(8,2) = "yes"
      let rule(8,3) = "no"     let rule(8,4) = "operating"
      let rule(9,1) = "stop"   let rule(9,2) = "yes"
      let rule(9,3) = "yes"    let rule(9,4) = "operating"
      let rule(10,1) = "none"  let rule(10,2) = "yes"
      let rule(10,3) = "no"    let rule(10,4) = "operating"
      let rule(11,1) = "none"  let rule(11,2) = "yes"
      let rule(11,3) = "yes"   let rule(11,4) = "operating"
      let rule(12,1) = "start" let rule(12,2) = "yes"
      let rule(12,3) = "no"    let rule(12,4) = "operating"
      let rule(13,1) = "start" let rule(13,2) = "yes"
      let rule(13,3) = "yes"   let rule(13,4) = "operating"
      let rule(14,1) = ""      let rule(14,2) = ""
      let rule(14,3) = ""      let rule(14,4) = "standby*"
      let rule(15,1) = ""      let rule(15,2) = "no"
      let rule(15,3) = ""      let rule(15,4) = "operating*"
      let rule(16,1) = ""      let rule(16,2) = "yes"
      let rule(16,3) = "no"    let rule(16,4) = "operating*"
      let rule(17,1) = ""      let rule(17,2) = "yes"
      let rule(17,3) = "yes"   let rule(17,4) = "operating*"
      let later.case = .yes
   always
''
''   Determine input signal status.  Assume that "start" and "stop"
''   commands cancel each other out (respective values of 1 and -1).
''
   for every signal in input.sset(component)
   do
      if signal.type(signal) eq "process"
         add 1 to total.process
         if strength(signal) eq .on
            add 1 to number.process
         always
      else
         if signal.type(signal) eq "power"
            add 1 to total.power
            if strength(signal) eq .on
               add 1 to number.power
            always
         else
            add 1 to total.command
            add strength(signal) to index.command
         always
      always
   loop
''
''   Develop test vector for comparison with rules.  Assume that
```

```
''   a single process signal is sufficient, and that a single power
''   signal is sufficient  (i.e., OR gates).
''
   if index.command eq -1
      let test(1) = "stop"
   else
      if index.command eq 0
         let test(1) = "none"
      else
         let test(1) = "start"
      always
   always
   if number.power ge 1
      let test(2) = "yes"
   else
      let test(2) = "no"
   always
   if number.process ge 1
      let test(3) = "yes"
   else
      let test(3) = "no"
   always
   let test(4) = state(component)
''
''   Determine appropriate rule.
''
   for ruletype = 1 to 17
   do
      for j = 1 to 4
      do
         if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
            go to 'next'
         always
      loop
      go to 'found'
   'next'
   loop
''
''   Select rule.
''
   'found'
   select case ruletype

   case 1, 16
      let state(component) = "failed"
      let output.strength = .no

   case 2, 3, 4, 7
      let state(component) = "standby"
      let output.strength = .no

   case 5
      call demand.test giving component yielding success
      if success eq .no
         let state(component) = "standby*"
```

```
         let output.strength = .no
      else
         let state(component) = "failed"
         let output.strength = .no
      always

   case 6
      call demand.test giving component yielding success
      if success eq .no
         let state(component) = "standby*"
         let output.strength = .no
      else
         let state(component) = "operating"
         let output.strength = .yes
      always

   case 8
      call demand.test giving component yielding success
      if success eq .no
         let state(component) = "failed"
         let output.strength = .no
      else
         let state(component) = "standby"
         let output.strength = .no
      always

   case 9
      call demand.test giving component yielding success
      if success eq .no
         let state(component) = "operating*"
         let output.strength = .yes
      else
         let state(component) = "standby"
         let output.strength = .no
      always

   case 10, 12
      let state(component) = "failed"
      let output.strength = .no

   case 11, 13
      let state(component) = "operating"
      let output.strength = .yes

   case 14
      let state(component) = "standby*"
      let output.strength = .no

   case 15
      let state(component) = "operating*"
      let output.strength = .no

   case 17
      let state(component) = "operating*"
      let output.strength = .yes
```

```
    default
''
''    Error messages can be put here if rule not matched.
''
    endselect
''
''    Update output signals.
''
    for every signal in output.sset(component)
       let strength(signal) = output.strength

    return

end ''active
```

```
process availability
''
''     This process totals the sum of the system indicator
''     variable at the specified time points.  At the completion
''     of all trials the totals are divided by the number of
''     trials to determine the time dependent system availability.
''
   while time.v lt (simulation.time + 10)
   do
      suspend
      add system.ind.var to time.avail.data(availability)
   loop

   suspend

end ''availability
```

```
process call.update
''
''   This should be a process to keep the process component
''   from destroying itself when it tries to call a system
''   update.
''
   while time.v lt .000004
   do
      wait .000005 hours
      for every component in system.cset
      do
         resume the component
      loop
      for every tank in system.tset                                    ''TANK
      resume the tank                                                  ''TANK
      wait .0005 hours
      for i = 1 to dim.f(cptr(*))
      do
         if component.type(cptr(i)) eq "active"
            or component.type(cptr(i)) eq "passive"
            if state(cptr(i)) ne "operating"
               interrupt the component called cptr(i)
            always
         always
      loop
   loop
   call system.update

   return

end ''call.update
```

```
routine check.valve given component
''
''   Develops output signals for a check valve.
''                               ________
''                              |        |
''        input process      ---|        |--- output process
''                              |________|
''
''   Condensed decision table:
''
''          Process    Initial          Final        Process
''    Case   Input      State           State        Output
''    ----  -------     -----           -----        ------
''      1      -     failed_closed   failed_closed    no
''      2     no        closed          closed        no
''      3     yes       closed       failed_closed    no
''                                      open          yes
''      4     no      failed_open     failed_open     no
''      5     yes     failed_open     failed_open     yes
''      6     no         open         failed_open     no
''                                      closed        no
''      7     yes        open           open          yes
''
   define rule as a saved 2-dimensional text array
   define component as a pointer variable
   define number.process, total.process, output.strength,
      ruletype, success and j as integer variables
   define later.case as a saved integer variable
''
''   Enter decision table.
''
   if later.case eq .no
      reserve rule as 7 by 2
      let rule(1,1) = ""         let rule(1,2) = "failed_closed"
      let rule(2,1) = "no"       let rule(2,2) = "closed"
      let rule(3,1) = "yes"      let rule(3,2) = "closed"
      let rule(4,1) = "no"       let rule(4,2) = "failed_open"
      let rule(5,1) = "yes"      let rule(5,2) = "failed_open"
      let rule(6,1) = "no"       let rule(6,2) = "open"
      let rule(7,1) = "yes"      let rule(7,2) = "open"
      let later.case = .yes
   always
''
''   Determine input signal status.
''
   for every signal in input.sset(component)
   do
      if signal.type(signal) eq "process"
         add 1 to total.process
         if strength(signal) eq .on
            add 1 to number.process
         always
      always
   loop
''
```

```
''   Develop test vector for comparison with rules.  Assume that
''   a single process signal is sufficient (i.e., an OR gate).
''
    if number.process ge 1
       let test(1) = "yes"
    else
       let test(1) = "no"
    always
    let test(2) = state(component)
''
''   Determine appropriate rule.
''
    for ruletype = 1 to 7
    do
       for j = 1 to 2
       do
          if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
             go to 'next'
          always
       loop
       go to 'found'
    'next'
    loop
''
''   Select rule.
''
    'found'
    select case ruletype

    case 1
       let state(component) = "failed_closed"
       let output.strength = .no

    case 2
       let state(component) = "closed"
       let output.strength = .no

    case 3
       call demand.test giving component yielding success
       if success eq .no
          let state(component) = "failed_closed"
          let output.strength = .no
       else
          let state(component) = "open"
          let output.strength = .yes
       always

    case 4
       let state(component) = "failed_open"
       let output.strength = .no

    case 5
       let state(component) = "failed_open"
       let output.strength = .yes

```

```
   case 6
      call demand.test giving component yielding success
      if success eq .no
         let state(component) = "failed_open"
         let output.strength = .no
      else
         let state(component) = "closed"
         let output.strength = .no
      always

   case 7
      let state(component) = "open"
      let output.strength = .yes

   default
''
''   Error messages can be put here if rule not matched.
''
   endselect
''
''   Update output signals.
''
   for every signal in output.sset(component)
      let strength(signal) = output.strength

   return

end ''check.valve
```

```
process component
''
''   Tracks behavior of all components after initial demand (change).
''   Includes repair.  Uses exponential failure time model.
''
   define mean.failure.time, default.time, e1, and
      e2 as real variables

   'term'
   suspend
   while time.v lt (simulation.time + 10)
   do
      'reset'
      let status(component) = .working
      if run.failure.frequency(component) gt 0
         let mean.failure.time = 1./run.failure.frequency(component)
         if run.type eq 1
            wait mean.failure.time hours
            go to 'repair'
         otherwise
         wait exponential.f(mean.failure.time,1) hours
         'repair'
         if status(component) eq .resetting
            go to 'reset'
         always
         if status(component) eq .reset.run
            go to 'term'
         always
         if state(component) eq "open" or
            state(component) eq "closed" or
            state(component) eq "operating"
            let old.state(component) = state(component)
            let state(component) = failure.translation(component)
            activate a call.update now
         always
      else
         let default.time = simulation.time + 10.0
         wait default.time hours
         if status(component) eq .resetting
            go to 'reset'
         always
         if status(component) eq .reset.run
            go to 'term'
         always
      always
         let status(component) = .awaiting.repair
         let failure.time(component) = time.v
         activate a repair.supervisor now
      suspend
      if status(component) eq .reset.run
         go to 'term'
      always
''
''   REPAIR
''
```

```
        let status(component) = .under.repair
        let e1 = repair.function.shape(component)
        let e2 = repair.function.scale(component)
        if run.type eq 1
           wait e2 hours
           go to 'good'
        otherwise
        wait weibull.f(e1,e2,1) hours
        'good'
        if status(component) eq .reset.run
           go to 'term'
        always
        let old.state(component) = state(component)
        select case component.type(component)

        case "active", "passive"
           let state(component) = "standby"

        case "switch"
           let state(component) = "open"

        case "valve", "check.valve"
           let state(component) = "closed"

        default
           print 1 line thus
           The component type was not matched in the repair routine.

        endselect
        activate a call.update now
     loop

     suspend

end ''component
```

```
routine demand.test given component yielding success
''
''   Determines if given component succeeds or fails on demand,
''   using the demand.failure.frequency for the component.
''
   define component as a pointer variable
   define success as an integer variable
   if random.f(1) le demand.failure.frequency(component)
      let success = .no
   else
      let success = .yes
   always

   return

end ''demand.test
```

```
process external.event
''
''   Schedules a change in the system (either to component status
''   or signal strength) occurrence.time hours into the simulation.
''
   while time.v lt (simulation.time + 10)
   do
      suspend
      for every component in extevnt.cset(external.event)
      do
         let old.state(component) = state(component)
         let state(component) = new.state(external.event)
      loop

      if number.signals(external.event) eq 1
      for j = 1 to number.signals(external.event)
      do
         for every signal in system.sset
            with origin(signal) eq signal.origin(external.event)
            and destination(signal) eq
                signal.destination(external.event)
            and signal.type(signal) eq signal.typee(external.event)
         find the first case
         if found
            let old.strength(signal) = strength(signal)
            let strength(signal) = new.strength(external.event)
         always
      loop
      else
      if number.signals(external.event) ne 0
         print 1 line thus
         An external event was entered with more than one signal change.
      always
      always
      call system.update
   loop

   suspend

end ''external.event
```

```
function failure.translation(component)
''
''   Determines status of "failed" component.
''
   define component as an integer variable
   define mode as a text variable

   select case component.type(component)

   case "active", "passive"
      let mode = "failed"

   case "check.valve", "valve", "switch"
      if state(component) eq "open"
         let mode = "failed_closed"
      always
      if state(component) eq "closed"
         let mode = "failed_open"
      always
      if state(component) ne "open" and
         state(component) ne "closed"
         print 1 line thus
         Failure translation didn't function properly!
      always

   default
      print 1 line thus
      Failure translation routine rule not matched!

   endselect

   return with mode

end ''failure.translation
```

```
routine input
''
''   Problem input routine.
''
   define infile and outfile as text variables

   write as /, "Enter DOS input file name => ",+
   read infile
   write as /, "Enter DOS output file name => ",+
   read outfile
   open 7 for input, file name = infile
   use 7 for input
   open 8 for output, file name = outfile
   use 8 for output
''
''   Title, general characteristics.
''
   read job.title as t 80, /
   write job.title as t 80, /
   read simulation.time as d(10,2), /
   write simulation.time as d(10,2), /
   read run.type as i 10, /
   write run.type as i 10, /
   read ntrial as i 10, /
   write ntrial as i 10, /
   read ntimes as i 10, /
   write ntimes as i 10, /
   read distribution.type as i 10, /
   write distribution.type as i 10, /
   reserve time_avail(*) as ntimes
   if distribution.type eq 0
      for i = 1 to ntimes
      do
         read time_avail(i) as d(10,2), /
         write time_avail(i) as d(10,2), /
      loop
   always
''
''   Component characteristics.
''
   read n.component.record as i 10, /
   write n.component.record as i 10, /
   create every component.record
   reserve input.name(*,*), output.name(*,*), input.signal.type(*,*),
      output.signal.type(*,*), input.signal.strength(*,*), and
      output.signal.strength(*,*) as n.component.record by *
   for i = 1 to n.component.record
   do
      read component_name(i),
           component_type(i),
           initial_state(i),
           number_inputs(i), and
           number_outputs(i)
           as 3 t 10, 2 i 5, /
      write component_name(i),
```

```
            component_type(i),
            initial_state(i),
            number_inputs(i), and
            number_outputs(i)
            as 3 t 10, 2 i 5, /
       read demand_failure_frequency(i) and
            run_failure_frequency(i)
            as 2 d(10,5), /
       write demand_failure_frequency(i) and
            run_failure_frequency(i)
            as 2 d(10,5), /
       read repair_function_shape(i),
            repair_function_scale(i), and
            repair_probability(i)
            as 3 d(10,5), /
       write repair_function_shape(i),
            repair_function_scale(i), and
            repair_probability(i)
            as 3 d(10,5), /
''
''   Input signals for component.
''
       reserve input.name(i,*),
               input.signal.type(i,*), and
               input.signal.strength(i,*)
               as number_inputs(i)
       for j = 1 to number_inputs(i)
       do
          read input.name(i,j),
               input.signal.type(i,j), and
               input.signal.strength(i,j)
               as b 11, 2 t 10, i 5, /
          write input.name(i,j),
               input.signal.type(i,j), and
               input.signal.strength(i,j)
               as b 11, 2 t 10, i 5, /
          if trim.f(input.name(i,j),0) eq "system"
             add 1 to total.signal.count
          always
       loop
''
''   Output signals for components.
''
       reserve output.name(i,*),
               output.signal.type(i,*), and
               output.signal.strength(i,*)
               as number_outputs(i)
       for j = 1 to number_outputs(i)
       do
          read output.name(i,j),
               output.signal.type(i,j), and
               output.signal.strength(i,j)
               as b 11, 2 t 10, i 5, /
          write output.name(i,j),
               output.signal.type(i,j), and
```

```
              output.signal.strength(i,j)
              as b 11, 2 t 10, i 5, /
      loop
      add number_outputs(i) to total.signal.count
   loop
''
''   System characteristics.
''
   read initial.system.state as t 10, /
   write initial.system.state as t 10, /
   read system.success.criterion as i 10, /
   write system.success.criterion as i 10, /
''
''   External event records.
''
   read n.external.event.record as i 10, /
   write n.external.event.record as i 10, /
   if n.external.event.record gt 0
      create every external.event.record
      reserve extevnt.component(*,*), extevnt.origin(*,*), and
         extevnt.destination(*,*), extevnt.stype(*,*)
         as n.external.event.record by *

      for i = 1 to n.external.event.record
      do
         read occurrence_time(i) as d(10,2)
         write occurrence_time(i) as d(10,2)
         read number_components(i) as i 10, /
         write number_components(i) as i 10, /
         if number_components(i) gt 0
            reserve extevnt.component(i,*) as number_components(i)
            for j = 1 to number_components(i)
            do
               read extevnt.component(i,j) as t 10
               write extevnt.component(i,j) as t 10
            loop
            read new_state(i) as /, t 10, /
            write new_state(i) as /, t 10, /
         always
         read number_signals(i) as i 10, /
         write number_signals(i) as i 10, /
         if number_signals(i) gt 0
            reserve extevnt.origin(i,*), extevnt.destination(i,*),
            extevnt.stype(i,*) as number_signals(i)
            for j = 1 to number_signals(i)
            do
               read extevnt.origin(i,j),
                    extevnt.destination(i,j),
                    extevnt.stype(i,j)
                    as 3 t 10, /
               write extevnt.origin(i,j),
                    extevnt.destination(i,j),
                    extevnt.stype(i,j)
                    as 3 t 10, /
            loop
```

```
            read new_strength(i) as  i 10, /
            write new_strength(i) as  i 10, /
        always
     loop
  always

end ''input
```

```
routine passive given component
''
''   Develops output signals for a passive component (no explicit
''   command signals or power source).
''                                 _______
''                                |       |
''        input process        ---|       |--- output process
''                                |_______|
''
''   Condensed decision table:
''
''                Process  Initial     Final    Process
''       Case      Input    State      State    Output
''       ----     -------   -----      -----    ------
''         1         -      failed     failed     no
''         2        no      standby    standby    no
''         3        yes     standby    failed     no
''                                    operating   yes
''         4        no     operating   standby    no
''         5        yes    operating  operating   yes
''
   define rule as a saved 2-dimensional text array
   define component as a pointer variable
   define number.process, total.process, output.strength,
      ruletype, success, and j as integer variables
   define later.case as a saved integer variable
''
''   Enter decision table.
''
   if later.case eq .no
      reserve rule as 5 by 2
      let rule(1,1) = ""            let rule(1,2) = "failed"
      let rule(2,1) = "no"          let rule(2,2) = "standby"
      let rule(3,1) = "yes"         let rule(3,2) = "standby"
      let rule(4,1) = "no"          let rule(4,2) = "operating"
      let rule(5,1) = "yes"         let rule(5,2) = "operating"
      let later.case = .yes
   always
''
''   Determine input signal status.
''
   for every signal in input.sset(component)
   do
      if signal.type(signal) eq "process"
         add 1 to total.process
         if strength(signal) eq .on
            add 1 to number.process
         always
      always
   loop
''
''   Develop test vector for comparison with rules.  Assume that
''   a single process signal is sufficient (i.e., an OR gate).
''
   if number.process ge 1
```

```
      let test(1) = "yes"
   else
      let test(1) = "no"
   always
   let test(2) = state(component)
''
''   Determine appropriate rule.
''
   for ruletype = 1 to 5
   do
      for j = 1 to 2
      do
         if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
            go to 'next'
         always
      loop
      go to 'found'
   'next'
   loop
''
''   Select rule.
''
   'found'
   select case ruletype

   case 1
      let state(component) = "failed"
      let output.strength = .no

   case 2
      let state(component) = "standby"
      let output.strength = .no

   case 3
      call demand.test giving component yielding success
      if success eq .no
         let state(component) = "failed"
         let output.strength = .no
      else
         let state(component) = "operating"
         let output.strength = .yes
      always

   case 4
      let state(component) = "standby"
      let output.strength = .no

   case 5
      let state(component) = "operating"
      let output.strength = .yes

   default
''
''   Error messages can be put here if rule not matched.
''
```

```
   endselect
''
''   Update output signals.
''
   for every signal in output.sset(component)
      let strength(signal) = output.strength

   return

end ''passive
```

```
process repair.supervisor
''
''    This process can be modified in the future to determine
''    when a failed component should begin the repair process.
''    Time delays can be inserted (repair delays) and if repair
''    resources are limited the number of components under
''    repair at any given time can be controlled here.
''
''    Currently this routine will be called from the system.update
''    routine every time a new failure is detected.  This routine
''    uses the repair.probability for the failed component to
''    determine if the component is repairable or not.  If the
''    component is repairable a repair is then begun immediately.
''    To determine what the current status of each component is
''    the status variable can be checked.  The status will be
''    working, resetting, awaiting repair, under repair, or not
''    repairable.
''
''    This portion is for defining a repair delay.
''
   define component as a pointer variable
   define a, b, and x as real variables
   let a = 1.0
   let b = 100.0
   let x = time.v
   if run.type eq 1
      wait b hours
      let a = 0.0
      go to 'good'
   otherwise
''   wait weibull.f(a,b,1) hours
   'good'
''
''    If it is desireable to use various repair delays on a frequent
''    basis, the program could be modified to read in the repair
''    delay distribution parameters.  The above delay is a weibull
''    distribution, but with the parameters chosen, it is actually
''    an exponential distribution.
''
   for every component in system.cset
      with failure.time(component) eq x
   find the first case
   if found
      if status(component) = .awaiting.repair
         if random.f(1) le repair.probability(component)
            resume the component
         else
            let status(component) = .not.repairable
         always
      always
      let failure.time(component) = -1.0
   else
      print 1 line thus
      In repair supervisor routine the component to repair was not IDed.
   always
```

```
    return

end ''repair.supervisor
```

```
routine run.initialize
''
''   initialization of components, signals, and external events
''
   define i, j, k, and signal.count as integer variables
   define x, y, and z as real variables
''
''   Component initialization.
''
   reserve cptr(*) as n.component.record
   for i = 1 to n.component.record
   do
      activate a component called cptr(i) now
      file cptr(i) in system.cset
      let name(cptr(i)) = trim.f(component_name(i),0)
      let component.type(cptr(i)) = trim.f(component_type(i),0)
      let n.input.sset(cptr(i)) = number_inputs(i)
      let n.output.sset(cptr(i)) = number_outputs(i)
      let demand.failure.frequency(cptr(i)) =
          demand_failure_frequency(i)
      let run.failure.frequency(cptr(i)) = run_failure_frequency(i)
      let repair.probability(cptr(i)) = repair_probability(i)
      let repair.function.shape(cptr(i)) = repair_function_shape(i)
      let repair.function.scale(cptr(i)) = repair_function_scale(i)

      select case component.type(cptr(i))

      case "active"
         let response.function(cptr(i)) = 'active'

      case "passive"
         let response.function(cptr(i)) = 'passive'

      case "valve"
         let response.function(cptr(i)) = 'valve'

      case "check_valve"
         let response.function(cptr(i)) = 'check.valve'

      case "switch", "breaker"
         let response.function(cptr(i)) = 'switch'

      default
         let response.function(cptr(i)) = 'active'
         print 1 line with name(cptr(i)) thus
      In initialize routine response function not matched to *********

      endselect

   loop
   add 5 to total.signal.count                                       ''TANK
   reserve sptr(*) as total.signal.count
''
''   Initialize and file boundary condition signals.
''
```

```
    for j = 1 to n.component.record
    do
       for k = 1 to number_inputs(j)
       do
          if trim.f(input.name(j,k),0) eq "system"
             add 1 to signal.count
             create a signal called sptr(signal.count)
             let signal.type(sptr(signal.count)) =
                   trim.f(input.signal.type(j,k),0)
             let origin(sptr(signal.count)) = "system"
             let destination(sptr(signal.count)) =
                   trim.f(component_name(j),0)
             file sptr(signal.count) in input.sset(cptr(j))
             file sptr(signal.count) in system.boundary.sset
             file sptr(signal.count) in system.sset
          always
       loop
    loop
''
''  Initialize and file component output signals.
''
    for j = 1 to n.component.record
    do
       for k = 1 to number_outputs(j)
       do
          add 1 to signal.count
          create a signal called sptr(signal.count)
          let signal.type(sptr(signal.count)) =
               trim.f(output.signal.type(j,k),0)
          let origin(sptr(signal.count)) = trim.f(component_name(j),0)
          let destination(sptr(signal.count)) =
               trim.f(output.name(j,k),0)
          for every component in system.cset
             with name(component) eq destination(sptr(signal.count))
             find the first case
             if found
                file sptr(signal.count) in input.sset(component)
             else
                if destination(sptr(signal.count)) eq "system"
                   file sptr(signal.count) in system.success.sset
                always
             always
          file sptr(signal.count) in output.sset(cptr(j))
          file sptr(signal.count) in system.sset
       loop
    loop
''
''  Create and initialize external events, using
''  permanent entity external.event.record.
''
    if n.external.event.record gt 0
       reserve eptr(*) as n.external.event.record
       for i = 1 to n.external.event.record
       do
          activate an external.event called eptr(i) now
```

```
            let occurrence.time(eptr(i)) = occurrence_time(i)
            add .001 to occurrence.time(eptr(i))
            let new.state(eptr(i)) = trim.f(new_state(i),0)
            for j = 1 to number_components(i)
            do
               for every component in system.cset
                  with name(component) eq trim.f(extevnt.component(i,j),0)
                  find the first case
                  if found
                     file component in extevnt.cset(eptr(i))
                  always
            loop
            let new.strength(eptr(i)) = new_strength(i)
            let number.signals(eptr(i)) = number_signals(i)
            if number.signals(eptr(i)) eq 1
               let signal.origin(eptr(i)) = trim.f(extevnt.origin(i,1),0)
               let signal.destination(eptr(i)) =
                  trim.f(extevnt.destination(i,1),0)
               let signal.typee(eptr(i)) = trim.f(extevnt.stype(i,1),0)
            always
            file eptr(i) in system.eset
         loop
      always

      reserve test as 4
      reserve signal.status(*) as dim.f(sptr(*))
      reserve unavailability.dist(*) as ntrial
      reserve aptr(*) as ntimes
      if distribution.type eq 1
         let x = simulation.time / (ntimes - 1)
         let time_avail(1) = 0.
         for i = 2 to ntimes
         do
            let time_avail(i) = (i - 1) * x
         loop
      always
      if distribution.type eq 2
         let y = log.10.f(simulation.time)
         let x = y / (ntimes - 1)
         let time_avail(1) = 0.
         for i = 2 to ntimes
         do
            let z = (i - 1) * x
            let time_avail(i) = 10 ** z
         loop
      always
      for i = 1 to ntimes
      do
         activate an availability called aptr(i) now
         let time.avail(aptr(i)) = time_avail(i)
      loop

      return

   end ''run.initialize
```

```
routine run.output
''
''    This routine will print the output report at the end of the
''    run.  It prints the time dependent unavailability data and the
''    average unavailability distribution data.
''
   define x as a real variable

   for i = 1 to ntimes
   do
      let x = time.avail.data(aptr(i))
      let time.avail.data(aptr(i)) = x / ntrial
      let x = 1 - time.avail.data(aptr(i))
      let time.avail.data(aptr(i)) = x
   loop

   write as *,/,/
   print 6 lines with ntrial thus
                              AFTER **** TRIALS

                 THE TIME DEPENDENT UNAVAILABILITY IS AS FOLLOWS

                           TIME                UNAVAILABILITY
            ---------------------------------------------------------
   for i = 1 to ntimes
   do
      print 2 lines with time.avail(aptr(i))
         and time.avail.data(aptr(i)) thus

                         *****.**                   *.****
   loop
''
''   Sort the average unavailability distribution data.
''
   define l, m, n, j, k, and im as integer variables
   define xp as a real variable

   let m = ntrial
   'sort1'
   let l = m
   let m = div.f(l,2)
   if m gt 0
      let k = ntrial - m
      for j = 1 to k
      do
         let n = j
         'sort2'
         let im = n + m
         if unavailability.dist(n) gt unavailability.dist(im)
            let xp = unavailability.dist(n)
            let unavailability.dist(n) = unavailability.dist(im)
            let unavailability.dist(im) = xp
            let l = n
            let n = l - m
            if n gt 0
```

```
            go to 'sort2'
         otherwise
      always
   loop
   if m gt 0
      go to 'sort1'
   otherwise
always

write as *,/,/
print 6 lines with ntrial and simulation.time thus
                              AFTER **** TRIALS
                                     AND
                       OVER A TIME PERIOD OF ***** HOURS
                THE AVERAGE SYSTEM UNAVAILABILITY IS AS FOLLOWS
              ---------------------------------------------------

define x1, x5, x25, x40, x50, x60, x75, x95, and x99
   as integer variables
let x1 = div.f(ntrial,100)
let x = 5 * ntrial
let x5 = div.f(x,100)
let x = 25 * ntrial
let x25 = div.f(x,100)
let x = 40 * ntrial
let x40 = div.f(x,100)
let x50 = div.f(ntrial,2)
let x = 60 * ntrial
let x60 = div.f(x,100)
let x = 75 * ntrial
let x75 = div.f(x,100)
let x = 95 * ntrial
let x95 = div.f(x,100)
let x = 99 * ntrial
let x99 = div.f(x,100)
if x1 eq 0
   let x1 = 1
always
if x5 eq 0
   let x5 = 1
always
print 27 lines with minimum.unavailability, unavailability.dist(x1),
   unavailability.dist(x5), unavailability.dist(x25),
   unavailability.dist(x40), unavailability.dist(x50),
   unavailability.dist(x60),unavailability.dist(x75),
   unavailability.dist(x95), unavailability.dist(x99),
   maximum.unavailability, average.unavailability,
   and variance.unavailability thus

                        The minimum is:            *.****

                        The 1st percentile is:     *.****

                        The 5th percentile is:     *.****

```

```
                          The 25th percentile is:    *.****

                          The 40th percentile is:    *.****

                          The 50th percentile is:    *.****

                          The 60th percentile is:    *.****

                          The 75th percentile is:    *.****

                          The 95th percentile is:    *.****

                          The 99th percentile is:    *.****

                          The maximum is:            *.****

                          The mean is:               *.****

                          The variance is:           *.****

''
''     Use this portion to print out all of the average system
''     unavailability values, one for every trial.  These are the
''     values on which the above percentiles are based.
''
''    write as *,/,/
''    for i = 1 to ntrial
''    do
''       print 1 line with i and unavailability.dist(i) thus
''       point **** is *.****
''    loop

end ''run.output
```

```
process schedule.avail.samples
''
''   This process will cause samples to be taken at the designated
''   times during each trial to compute the time dependent
''   availability of the system.
''
    define x as a real variable

    wait .002 hours
    resume the availability called aptr(1)
    for i = 2 to ntimes
    do
       let x = time.avail(aptr(i)) - time.avail(aptr(i - 1))
       wait x hours
       resume the availability called aptr(i)
    loop

    return

end ''schedule.avail.samples
```

```
process schedule.external.events
''
''   Schedules external events.
''
   define i as an integer variable
   define x as a real variable

   if n.external.event.record gt 0
      wait occurrence.time(eptr(1)) hours
      resume the external.event called eptr(1)
      for i = 2 to dim.f(eptr(*))
      do
         let x = occurrence.time(eptr(i)) - occurrence.time(eptr(i - 1))
         wait x hours
         resume the external.event called eptr(i)
      loop
   always

   return

end ''schedule.external.events
```

```
process stop.scenario
''
''   This process will interrupt any external events or components
''   still scheduled to occur later in time.  It then resets all
''   components so they can begin operation again in the next trial.
''
   call system.update

   for every external.event in ev.s(i.external.event)
      interrupt external.event

   for every component in ev.s(i.component)
   do
      interrupt component
      let time.a(component) = 0.0
   loop

   for every component in system.cset
   do
      let status(component) = .reset.run
      resume component
   loop

   return

end ''stop.scenario
```

```
routine switch given component
''
''   Develops output signals for a switch or breaker
''   using explicit command signals.  Assumes that the component
''   has one or more command signal inputs, power inputs, and
''   process inputs:
''
''        input command  ---|______|
''        input power    ---|      |--- output process
''        input process  ---|______|
''
''   Condensed decision table:
''
''      Command   Power  Process      Initial        Final      Process
''Case   Input    Input   Input      State           State       Output
''----  -------   -----  -------     -----           -----       ------
''  1      -        -       -      failed_open     failed_open     no
''  2      -       no       -         open            open         no
''  3    open       -       -         open            open         no
''  4    none       -       -         open            open         no
''  5    close     yes     no         open         failed_open     no
''                                                   closed        no
''  6    close     yes     yes        open         failed_open     no
''                                                   closed        yes
''  7      -        -      no      failed_closed   failed_closed   no
''  8      -        -      yes     failed_closed   failed_closed   yes
''  9      -       no      no        closed          closed        no
'' 10      -       no      yes       closed          closed        yes
'' 11   open      yes     no        closed        failed_closed   no
''                                                    open         no
'' 12   open      yes     yes       closed        failed_closed   yes
''                                                    open         no
'' 13   none       -      no        closed          closed        no
'' 14   none       -      yes       closed          closed        yes
'' 15   close      -      no        closed          closed        no
'' 16   close      -      yes       closed          closed        yes
''
   define rule as a saved 2-dimensional text array
   define component as a pointer variable
   define index.command, total.command, number.power, total.power,
      number.process, total.process, output.strength, ruletype,
      success and j as integer variables
   define later.case as a saved integer variable
''
''   Enter decision table.
''
   if later.case eq .no
      reserve rule as 16 by 4
      let rule(1,1) = ""        let rule(1,2) = ""
      let rule(1,3) = ""        let rule(1,4) = "failed_open"
      let rule(2,1) = ""        let rule(2,2) = "no"
      let rule(2,3) = ""        let rule(2,4) = "open"
      let rule(3,1) = "open"    let rule(3,2) = ""
      let rule(3,3) = ""        let rule(3,4) = "open"
      let rule(4,1) = "none"    let rule(4,2) = ""
```

```
    let rule(4,3) = ""        let rule(4,4) = "open"
    let rule(5,1) = "close"   let rule(5,2) = "yes"
    let rule(5,3) = "no"      let rule(5,4) = "open"
    let rule(6,1) = "close"   let rule(6,2) = "yes"
    let rule(6,3) = "yes"     let rule(6,4) = "open"
    let rule(7,1) = ""        let rule(7,2) = ""
    let rule(7,3) = "no"      let rule(7,4) = "failed_closed"
    let rule(8,1) = ""        let rule(8,2) = ""
    let rule(8,3) = "yes"     let rule(8,4) = "failed_closed"
    let rule(9,1) = ""        let rule(9,2) = "no"
    let rule(9,3) = "no"      let rule(9,4) = "closed"
    let rule(10,1) = ""       let rule(10,2) = "no"
    let rule(10,3) = "yes"    let rule(10,4) = "closed"
    let rule(11,1) = "open"   let rule(11,2) = "yes"
    let rule(11,3) = "no"     let rule(11,4) = "closed"
    let rule(12,1) = "open"   let rule(12,2) = "yes"
    let rule(12,3) = "yes"    let rule(12,4) = "closed"
    let rule(13,1) = "none"   let rule(13,2) = ""
    let rule(13,3) = "no"     let rule(13,4) = "closed"
    let rule(14,1) = "none"   let rule(14,2) = ""
    let rule(14,3) = "yes"    let rule(14,4) = "closed"
    let rule(15,1) = "close"  let rule(15,2) = ""
    let rule(15,3) = "no"     let rule(15,4) = "closed"
    let rule(16,1) = "close"  let rule(16,2) = ""
    let rule(16,3) = "yes"    let rule(16,4) = "closed"
    let later.case = .yes
 always
''
''   Determine input signal status.  Assume that "open" and "close"
''   commands cancel each other out (respective values of 1 and -1).
''
 for every signal in input.sset(component)
 do
    if signal.type(signal) eq "process"
       add 1 to total.process
       if strength(signal) eq .on
          add 1 to number.process
       always
    else
       if signal.type(signal) eq "power"
          add 1 to total.power
          if strength(signal) eq .on
             add 1 to number.power
          always
       else
          add 1 to total.command
          add strength(signal) to index.command
       always
    always
 loop
''
''   Develop test vector for comparison with rules.  Assume that
''   a single process signal is sufficient, and that a single power
''   signal is sufficient  (i.e., OR gates).
''
```

```
if index.command eq -1
   let test(1) = "close"
else
   if index.command eq 0
      let test(1) = "none"
   else
      let test(1) = "open"
   always
always
if number.power ge 1
   let test(2) = "yes"
else
   let test(2) = "no"
always
if number.process ge 1
   let test(3) = "yes"
else
   let test(3) = "no"
always
let test(4) = state(component)
''
''  Determine appropriate rule.
''
for ruletype = 1 to 16
do
   for j = 1 to 4
   do
      if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
         go to 'next'
      always
   loop
   go to 'found'
'next'
loop
''
''  Select rule.
''
'found'
select case ruletype

case 1
   let state(component) = "failed_open"
   let output.strength = .no

case 2, 3, 4
   let state(component) = "open"
   let output.strength = .no

case 5
   call demand.test giving component yielding success
   if success eq .no
      let state(component) = "failed_open"
      let output.strength = .no
   else
      let state(component) = "closed"
```

```
            let output.strength = .no
        always

    case 6
        call demand.test giving component yielding success
        if success eq .no
            let state(component) = "failed_open"
            let output.strength = .no
        else
            let state(component) = "closed"
            let output.strength = .yes
        always

    case 7
        let state(component) = "failed_closed"
        let output.strength = .no

    case 8
        let state(component) = "failed_closed"
        let output.strength = .yes

    case 9, 13, 15
        let state(component) = "closed"
        let output.strength = .no

    case 10, 14, 16
        let state(component) = "closed"
        let output.strength = .yes

    case 11
        call demand.test giving component yielding success
        if success eq .no
            let state(component) = "failed_closed"
            let output.strength = .no
        else
            let state(component) = "open"
            let output.strength = .no
        always

    case 12
        call demand.test giving component yielding success
        if success eq .no
            let state(component) = "failed_closed"
            let output.strength = .yes
        else
            let state(component) = "open"
            let output.strength = .no
        always

    default
''
''   Error messages can be put here if rule not matched.
''
    endselect
''
```

```
''   Update output signals.
''
   for every signal in output.sset(component)
      let strength(signal) = output.strength

   return

end ''switch
```

```
routine system.update
''
''   Updates status of signals in system, given status of all components
''   Performs iterations until signals stabilize or number of iterations
''   is exceeded.
''
''   Notes:
''   1)  Currently, maximum is set by number of signals.  Later
''       versions might make use of digraph/Petri net results.
''   2)  Current version re-analyzes every component.  Later versions
''       might only re-analyze components whose input changes.
''
   define rf as a subprogram variable
   define i, itr, max.itr and number.success
      as integer variables

   for i = 1 to dim.f(sptr(*))
      let signal.status(i) = strength(sptr(i))

   let max.itr = dim.f(sptr(*))
   for itr = 1 to max.itr
   do
''
''    1)  Check for changed component states and changed input
''        signals.
''    2)  If found, place a demand on the component, and determine
''        component response. (Later versions may activate signals
''        here).  Note that since output signals are updated
''        in routine response.function, input signals for
''        downstream components are also updated.
''
      for every component in system.cset
      do
         if state(component) ne old.state(component)
            let rf = response.function(component)
            call rf giving component
         always
         for every signal in input.sset(component)
            with strength(signal) ne old.strength(signal)
            find the first case
            if found
               let rf = response.function(component)
               call rf giving component
            always
      loop
''
''    Quit iteration if no changes to entire set of signals.
''
      for i = 1 to dim.f(sptr(*))
         with strength(sptr(i)) ne signal.status(i)
         find the first case
         if found
            for i = 1 to dim.f(sptr(*))
               let signal.status(i) = strength(sptr(i))
         else
```

```
            go to 'update'
         always
   loop
   print 2 lines with 24*time.v thus
   !!! Error:  Iteration maximum exceeded in routine system.update
                            time = ****.*** hours.
''
''   Activate newly started components, interrupt newly stopped
''   components.
''
   'update'
   for every component in system.cset
   do
      if status(component) eq .working
         if state(component) ne old.state(component)
            select case component.type(component)

            case "active", "passive"
               if state(component) eq "failed"
                  or state(component) eq "standby*"
                  or state(component) eq "operating*"
                  if old.state(component) eq "operating"
                     interrupt the component
                  always
                  let time.a(component) = 0.0
                  resume the component
               always
               if state(component) eq "standby"
                  and old.state(component) eq "operating"
                  interrupt the component
               always
               if state(component) eq "operating"
                  and old.state(component) eq "standby"
                  let time.a(component) = 0.0
                  let status(component) = .resetting
                  resume the component
               always

            case "check.valve", "switch", "valve"
               if state(component) eq "closed"
                  and old.state(component) eq "open"
                  let status(component) = .resetting
                  interrupt the component
                  let time.a(component) = 0.0
                  resume the component
               always
               if state(component) eq "open"
                  and old.state(component) eq "closed"
                  let status(component) = .resetting
                  interrupt the component
                  let time.a(component) = 0.0
                  resume the component
               always
               if state(component) eq "failed_open"
                  or state(component) eq "failed_closed"
```

```
                 interrupt the component
                 let time.a(component) = 0.0
                 resume the component
              always

           default
           print 1 line thus
           When performing the system.update, no matching case!

           endselect
        always
     always
   loop
''
''   Update status of system, components and signals.
''
   for every signal in system.success.sset
   do
      if strength(signal) eq .on
         add 1 to number.success
      always
   loop
   if number.success ge system.success.criterion
      let system.state = "good"
      let system.ind.var = 1
   else
      let system.state = "failed"
      let system.ind.var = 0
   always

   call flow.update giving tptr(1)                                  ''TANK

   for every component in system.cset
      let old.state(component) = state(component)

   for every signal in system.sset
      let old.strength(signal) = strength(signal)

   return

end ''system.update
```

```
routine trial.initialize
''
''    This routine initializes the state of each component
''    and the strength of each signal at the beginning of
''    a trial.
''
   define i, j, and k as integer variables

   let system.state = trim.f(initial.system.state,0)
   if system.state eq "operating"
      let system.ind.var = 1
   else
      let system.ind.var = 0
   always
''
''    Component state initialization.
''
   for i = 1 to n.component.record
   do
      let old.state(cptr(i)) = trim.f(initial_state(i),0)
      let state(cptr(i)) = old.state(cptr(i))
   loop
''
''    Signal strength initialization.
''
   for i = 1 to n.component.record
   do
      for j = 1 to number_inputs(i)
      do
         for every signal in system.sset
            with origin(signal) eq "system"
            and destination(signal) eq trim.f(component_name(i),0)
            and signal.type(signal) eq trim.f(input.signal.type(i,j),0)
         find the first case
         if found
            let strength(signal) = input.signal.strength(i,j)
         always
      loop
      for k = 1 to number_outputs(i)
      do
         for every signal in system.sset
            with origin(signal) eq trim.f(component_name(i),0)
            and destination(signal) eq trim.f(output.name(i,k),0)
            and signal.type(signal) eq trim.f(output.signal.type(i,k),0)
         find the first case
         if found
            let strength(signal) = output.signal.strength(i,k)
         always
      loop
   loop

   return

end ''trial.initialize
```

```
routine valve given component
''
''   Develops output signals for an MOV or manual valve
''   using explicit command signals.  Assumes that the component
''   has one or more command signal inputs, power inputs, and
''   process inputs:
''                                  _______
''          input command   ---|       |
''          input power     ---|       |--- output process
''          input process   ---|_______|
''
''   Condensed decision table:
''
''         Command  Power  Process   Initial          Final     Process
''   Case   Input   Input   Input     State           State      Output
''   ----  -------  -----  -------    -----           -----     ------
''    1       -       -       -   failed_closed   failed_closed   no
''    2       -      no       -       closed          closed      no
''    3     close     -       -       closed          closed      no
''    4     none      -       -       closed          closed      no
''    5     open     yes     no       closed      failed_closed   no
''                                                    open        no
''    6     open     yes     yes      closed      failed_closed   no
''                                                    open        yes
''    7       -       -      no     failed_open    failed_open    no
''    8       -       -      yes    failed_open    failed_open    yes
''    9       -      no      no        open            open       no
''   10       -      no      yes       open            open       yes
''   11     close    yes     no        open        failed_open    no
''                                                    closed      no
''   12     close    yes     yes       open        failed_open    yes
''                                                    closed      no
''   13     none      -      no        open            open       ho
''   14     none      -      yes       open            open       yes
''   15     open      -      no        open            open       no
''   16     open      -      yes       open            open       yes
''
    define rule as a saved 2-dimensional text array
    define component as a pointer variable
    define index.command, total.command, number.power, total.power,
       number.process, total.process, output.strength, ruletype,
       success and j as integer variables
    define later.case as a saved integer variable
''
''   Enter decision table.
''
    if later.case eq .no
       reserve rule as 16 by 4
       let rule(1,1) = ""       let rule(1,2) = ""
       let rule(1,3) = ""       let rule(1,4) = "failed_closed"
       let rule(2,1) = ""       let rule(2,2) = "no"
       let rule(2,3) = ""       let rule(2,4) = "closed"
       let rule(3,1) = "close"  let rule(3,2) = ""
       let rule(3,3) = ""       let rule(3,4) = "closed"
       let rule(4,1) = "none"   let rule(4,2) = ""
```

```
      let rule(4,3) = ""        let rule(4,4) = "closed"
      let rule(5,1) = "open"    let rule(5,2) = "yes"
      let rule(5,3) = "no".     let rule(5,4) = "closed"
      let rule(6,1) = "open"    let rule(6,2) = "yes"
      let rule(6,3) = "yes"     let rule(6,4) = "closed"
      let rule(7,1) = ""        let rule(7,2) = ""
      let rule(7,3) = "no"      let rule(7,4) = "failed_open"
      let rule(8,1) = ""        let rule(8,2) = ""
      let rule(8,3) = "yes"     let rule(8,4) = "failed_open"
      let rule(9,1) = ""        let rule(9,2) = "no"
      let rule(9,3) = "no"      let rule(9,4) = "open"
      let rule(10,1) = ""       let rule(10,2) = "no"
      let rule(10,3) = "yes"    let rule(10,4) = "open"
      let rule(11,1) = "close"  let rule(11,2) = "yes"
      let rule(11,3) = "no"     let rule(11,4) = "open"
      let rule(12,1) = "close"  let rule(12,2) = "yes"
      let rule(12,3) = "yes"    let rule(12,4) = "open"
      let rule(13,1) = "none"   let rule(13,2) = ""
      let rule(13,3) = "no"     let rule(13,4) = "open"
      let rule(14,1) = "none"   let rule(14,2) = ""
      let rule(14,3) = "yes"    let rule(14,4) = "open"
      let rule(15,1) = "open"   let rule(15,2) = ""
      let rule(15,3) = "no"     let rule(15,4) = "open"
      let rule(16,1) = "open"   let rule(16,2) = ""
      let rule(16,3) = "yes"    let rule(16,4) = "open"
      let later.case = .yes
   always
''
''   Determine input signal status.  Assume that "open" and "close"
''   commands cancel each other out (respective values of 1 and -1).
''
   for every signal in input.sset(component)
   do
      if signal.type(signal) eq "process"
         add 1 to total.process
         if strength(signal) eq .on
            add 1 to number.process
         always
      else
         if signal.type(signal) eq "power"
            add 1 to total.power
            if strength(signal) eq .on
               add 1 to number.power
            always
         else
            add 1 to total.command
            add strength(signal) to index.command
         always
      always
   loop
''
''   Develop test vector for comparison with rules.  Assume that
''   a single process signal is sufficient, and that a single power
''   signal is sufficient  (i.e., OR gates).
''
```

```
    if index.command eq -1
       let test(1) = "close"
    else
       if index.command eq 0
          let test(1) = "none"
       else
          let test(1) = "open"
       always
    always
    if number.power ge 1
       let test(2) = "yes"
    else
       let test(2) = "no"
    always
''
''     By changing the test for number of process inputs, it is
''     possible to simulate k-out-of-n components.
''
    if number.process ge 1
       let test(3) = "yes"
    else
       let test(3) = "no"
    always
    let test(4) = state(component)
''
''   Determine appropriate rule.
''
    for ruletype = 1 to 16
    do
       for j = 1 to 4
       do
          if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
             go to 'next'
          always
       loop
       go to 'found'
    'next'
    loop
''
''   Select rule.
''
    'found'
    select case ruletype

    case 1
       let state(component) = "failed_closed"
       let output.strength = .no

    case 2, 3, 4
       let state(component) = "closed"
       let output.strength = .no

    case 5
       call demand.test giving component yielding success
       if success eq .no
```

```
         let state(component) = "failed_closed"
         let output.strength = .no
      else
         let state(component) = "open"
         let output.strength = .no
      always

   case 6
      call demand.test giving component yielding success
      if success eq .no
         let state(component) = "failed_closed"
         let output.strength = .no
      else
         let state(component) = "open"
         let output.strength = .yes
      always

   case 7
      let state(component) = "failed_open"
      let output.strength = .no

   case 8
      let state(component) = "failed_open"
      let output.strength = .yes

   case 9, 13, 15
      let state(component) = "open"
      let output.strength = .no

   case 10, 14, 16
      let state(component) = "open"
      let output.strength = .yes

   case 11
      call demand.test giving component yielding success
      if success eq .no
         let state(component) = "failed_open"
         let output.strength = .no
      else
         let state(component) = "closed"
         let output.strength = .no
      always

   case 12
      call demand.test giving component yielding success
      if success eq .no
         let state(component) = "failed_open"
         let output.strength = .yes
      else
         let state(component) = "closed"
         let output.strength = .no
      always

   default
''
```

```
''   Error messages can be put here if rule not matched.
''
   endselect
''
''   Update output signals.
''
   for every signal in output.sset(component)
      let strength(signal) = output.strength
   return
end ''valve
```

Appendix C
TANK Program Listing

```
routine flow.update given tank
''
''    Determine the new flow rate if it has changed.
''
    define tank as a pointer variable
    let flow.rate.in(tank) = 0
    let flow.rate.out(tank) = 0
    for every component in tank.input.cset(tank)
    do
        if name(component) eq "unit2"
            if state(component) eq "open"
                or state(component) eq "failed_open"
                    add 0.01 to flow.rate.in(tank)
            always
        else
            if name(component) eq "unit3"
                if state(component) eq "open"
                    or state(component) eq "failed_open"
                        add 0.005 to flow.rate.in(tank)
                always
            always
        always
    loop
    for every component in tank.output.cset(tank)
    do
        if state(component) eq "open"
            or state(component) eq "failed_open"
                add 0.01 to flow.rate.out(tank)
        always
    loop
    return
end ''flow.update
```

```
process stop.tank
''
''    This process will reset the tank process so it is ready
''    for the execution of another trial.
''
   for every tank in ev.s(i.tank)
   do
      interrupt the tank
   loop

   for every tank in system.tset
   do
      let level(tank) = 100.0
      let time.a(tank) = 0.0
      resume the tank
   loop

   return

end ''stop.tank
```

```
process tank
''
''    This routine will continuously monitor the water level
''    in a tank.
''
   'tankreset'
   suspend
   while time.v lt (simulation.time + 10)
   do
''
''    This portion of the routine determines if the tank is in the
''    proper control region and calls the tank update routine to
''    make changes if necessary.
''
   work continuously evaluating 'water.level' testing 'tank.condition'
   let net.flow.rate(tank) = flow.rate.in(tank) - flow.rate.out(tank)
   if level(tank) gt 90.0
      go to 'tankreset'
   otherwise
   call tank.update giving tank
   if level(tank) gt high.level(tank)
      or level(tank) lt low.level(tank)
         suspend
         go to 'tankreset'
   always

   loop

   suspend

end ''tank
```

```
function tank.condition(tank)
''
''    This function will cause calling of the tank update
''    routine if the tank status is not satisfactory.
''
   define tank as a pointer variable
''
''    Use this method to adjust tank flow rate only at the
''    end of integration time steps.
''
   define x as a real variable
   let x = flow.rate.in(tank) - flow.rate.out(tank)
   if net.flow.rate(tank) ne x
      return with 1
   otherwise
''
''    Is the tank too full?
''
   if level(tank) gt high.level(tank)
      return with 1
   otherwise
''
''    Is the tank too empty?
''
   if level(tank) lt low.level(tank)
      return with 1
   otherwise
''
''    Is the tank level high and the control state wrong?
''
   if level(tank) gt high.set(tank)
      for every component in system.cset
      do
         if name(component) eq "unit1"
            and state(component) eq "closed"
               return with 1
         otherwise
         if name(component) eq "unit2"
            and state(component) eq "open"
               return with 1
         otherwise
         if name(component) eq "unit3"
            and state(component) eq "open"
               return with 1
         otherwise
      loop
   always
''
''    Is the tank level low and the control state wrong?
''
   if level(tank) lt low.set(tank)
      for every component in system.cset
      do
         if name(component) eq "unit1"
            and state(component) eq "open"
```

```
                 return with 1
             otherwise
             if name(component) eq "unit2"
                and state(component) eq "closed"
                   return with 1
             otherwise
             if name(component) eq "unit3"
                and state(component) eq "closed"
                   return with 1
             otherwise
          loop
       always
''
''     Is the tank level satisfactory and the control state wrong?
''
    if level(tank) le high.set(tank)
       and level(tank) ge low.set(tank)
       for every component in system.cset
       do
          if name(component) eq "unit1"
             and state(component) eq "closed"
                return with 1
          otherwise
          if name(component) eq "unit2"
             and state(component) eq "closed"
                return with 1
          otherwise
          if name(component) eq "unit3"
             and state(component) eq "open"
                return with 1
          otherwise
       loop
    always
          return with 0

end ''tank.condition
```

```
routine tank.initialize.run
''
''    This routine initializes all of the variables associated
''    with the Aldemir Tank Problem.  Initializes for the number
''    of trials to be performed.
''
   define signal.count as an integer variable
   let integrator.v = 'runge.kutta.r'
   let max.step.v = 0.04166666666667      '' Approximately 1 hour
   let min.step.v = 0.04166666666667      '' Approximately 1 hour
   let abs.err.v = 0.001
   let rel.err.v = 0.1
''
''    Create a tank.
''
   reserve tptr(*) as 1
   activate a tank called tptr(1) now
   file tptr(1) in system.tset
   let high.level(tptr(1)) = 3.0
   let low.level(tptr(1)) = -3.0
   let high.set(tptr(1)) = 1.0
   let low.set(tptr(1)) = -1.0
''
''    Must create all of the Tank output signals since the base
''    program does not recognize the tank as a component.  These
''    signals include three command signals (one to each valve),
''    the tank process output to the outlet valve, and the process
''    output signal to the system for system status checking.
''
   let signal.count = 9
   create a signal called sptr(signal.count)
   let signal.type(sptr(signal.count)) = "command"
   let origin(sptr(signal.count)) = "tank"
   let destination(sptr(signal.count)) = "unit1"
      for every component in system.cset
         with name(component) eq "unit1"
      find the first case
      if found
      file sptr(signal.count) in input.sset(component)
      always
   file sptr(signal.count) in tank.output.sset(tptr(1))
   file sptr(signal.count) in system.sset
''
   add 1 to signal.count
   create a signal called sptr(signal.count)
   let signal.type(sptr(signal.count)) = "command"
   let origin(sptr(signal.count)) = "tank"
   let destination(sptr(signal.count)) = "unit2"
      for every component in system.cset
         with name(component) eq "unit2"
      find the first case
      if found
      file sptr(signal.count) in input.sset(component)
      always
   file sptr(signal.count) in tank.output.sset(tptr(1))
```

```
   file sptr(signal.count) in system.sset
''
   add 1 to signal.count
   create a signal called sptr(signal.count)
   let signal.type(sptr(signal.count)) = "command"
   let origin(sptr(signal.count)) = "tank"
   let destination(sptr(signal.count)) = "unit3"
      for every component in system.cset
         with name(component) eq "unit3"
      find the first case
      if found
      file sptr(signal.count) in input.sset(component)
      always
   file sptr(signal.count) in tank.output.sset(tptr(1))
   file sptr(signal.count) in system.sset
''
   add 1 to signal.count
   create a signal called sptr(signal.count)
   let signal.type(sptr(signal.count)) = "process"
   let origin(sptr(signal.count)) = "tank"
   let destination(sptr(signal.count)) = "unit1"
      for every component in system.cset
         with name(component) eq "unit1"
      find the first case
      if found
      file sptr(signal.count) in input.sset(component)
      always
   file sptr(signal.count) in tank.output.sset(tptr(1))
   file sptr(signal.count) in system.sset
''
   add 1 to signal.count
   create a signal called sptr(signal.count)
   let signal.type(sptr(signal.count)) = "process"
   let origin(sptr(signal.count)) = "tank"
   let destination(sptr(signal.count)) = "system"
   file sptr(signal.count) in tank.output.sset(tptr(1))
   file sptr(signal.count) in system.sset
   file sptr(signal.count) in system.success.sset
   for every component in system.cset
   do
      for every signal in output.sset(component)
      do
         if destination(signal) eq "tank"
            file signal in tank.input.sset(tptr(1))
            file component in tank.input.cset(tptr(1))
         always
      loop
      for every signal in input.sset(component)
         with signal.type(signal) eq "process"
      do
         if origin(signal) eq "tank"
            file component in tank.output.cset(tptr(1))
         always
      loop
   loop
```

```
    return

end ''tank.initialize.run
```

```
routine tank.initialize.trial
''
''    This routine will reset the appropriate values to begin
''    a new trial with the tank operating correctly.
''
   let level(tptr(1)) = 0.0
   let net.flow.rate(tptr(1)) = 0.0
   for every signal in tank.output.sset(tptr(1))
   do
''
''    Turn on the flow output and test signal from the tank.
''
      if signal.type(signal) = "process"
         let strength(signal) = .on
      always
''
''    Turn off the command signals for the valves to change position.
''
      if signal.type(signal) = "command"
         let strength(signal) = .off
      always
   loop

   return

end ''tank.initialize.trial
```

```
routine tank.update given tank
''
''    This routine determines the flow going in and out of the
''    tank and controls the opening and closing of the inlet and
''    outlet valves.  If the tank should happen to dryout or over
''    flow this routine will suspend the tank routine.
''
   define tank as a pointer variable
''
''    This is to track dryout.
''
   if level(tank) lt low.level(tank)
''       for every signal in tank.output.sset(tank)
''          with signal.type(signal) eq "process"
''       do
''          let strength(signal) = .no
''       loop
      go to 'leave'
   otherwise
''
''    This is to track overflow.
''
   if level(tank) gt high.level(tank)
      for every signal in tank.output.sset(tank)
         with destination(signal) eq "system"
      do
        let strength(signal) = .no
      loop
      go to 'leave'
   otherwise
   if level(tank) lt low.set(tank)
''
''    Close the outlet valve and open both inlet valves.
''
      for every component in tank.output.cset(tank)
      do
         for every signal in input.sset(component)
            with signal.type(signal) eq "command"
         do
           let strength(signal) = -1
         loop
      loop
      for every component in tank.input.cset(tank)
      do
         for every signal in input.sset(component)
            with signal.type(signal) eq "command"
         do
            let strength(signal) = 1
         loop
      loop
      go to 'leave'
   otherwise
   if level(tank) gt high.set(tank)
''
''    Open the outlet valve and close both inlet valves.
```

```
''
      for every component in tank.output.cset(tank)
      do
         for every signal in input.sset(component)
            with signal.type(signal) eq "command"
         do
            let strength(signal) = 1
         loop
      loop
      for every component in tank.input.cset(tank)
      do
         for every signal in input.sset(component)
            with signal.type(signal) eq "command"
         do
            let strength(signal) = -1
         loop
      loop
      go to 'leave'
   otherwise
''
''    If the level of the tank is in the operating range,
''    open the outlet valve(unit1) and the inlet valve from
''    unit2, but close the inlet valve from unit3.
''
   for every component in tank.output.cset(tank)
   do
      for every signal in input.sset(component)
         with signal.type(signal) eq "command"
      do
         let strength(signal) = 1
      loop
   loop
   for every component in tank.input.cset(tank)
   do
      if name(component) eq "unit2"
         for every signal in input.sset(component)
            with signal.type(signal) eq "command"
         do
            let strength(signal) = 1
         loop
      else
         if name(component) eq "unit3"
            for every signal in input.sset(component)
               with signal.type(signal) eq "command"
            do
               let strength(signal) = -1
            loop
         always
      always
   loop
   'leave'
   call system.update
   return

end ''tank.update
```

```
routine water.level(tank)
''
''    This routine supplies the integration rule for the continuous
''    variable level of the tank.
''
   define tank as a pointer variable
      let d.level(tank) = net.flow.rate(tank)*1440.0
''
''    We have left the time step as days and are reading flow rates
''    as meter level change per minute thus the factor of 1440 above.
''
end ''water.level
```

Appendix D
Sample Input Files

```
SINGLE COMPONENT, EXP REPAIR AND FAILURE, DUAL REPAIR STATES
  10000.00                                Time of simulation
         0                                Type of run (0 for normal)
       100                                Number of trials
        21                                Number of time points
         1                                Type of time distribution
         1                                Number of components
COMPONENT passive   operating     1    1  Component one
       0.0      0.01                      Failure data
       1.0      100.0       1.0           Repair data
          system    process       1       Input signal
          system    process       1       Output signal
standby                                   Initial system state
         1                                System success criteria
         0                                Number of external events
```

```
TWO OUT OF THREE PUMPS, EXPONENTIAL FAILURE AND REPAIR.
  10000.00                                        Time of simulation
         0                                        Type of run (0 for normal)
       100                                        Number of trials
        21                                        Number of time points
         1                                        Type of time distribution
         4                                        Number of components
PUMP1     active    operating    3    1   Component one
       0.0      0.01                      Failure data
       1.0      100.0      1.0            Repair data
           system    power        1       Input signal
           system    command      1       Input signal
           system    process      1       Input signal
           VALVE     process      1       Output signal
PUMP2     active    operating    3    1   Component two
       0.0      0.01                      Failure data
       1.0      100.0      1.0            Repair data
           system    power        1       Input signal
           system    command      1       Input signal
           system    process      1       Input signal
           VALVE     process      1       Output signal
PUMP3     active    operating    3    1   Component three
       0.0      0.01                      Failure data
       1.0      100.0      1.0            Repair data
           system    power        1       Input signal
           system    command      1       Input signal
           system    process      1       Input signal
           VALVE     process      1       Output signal
VALVE     valve     open         5    1   Component four
       0.0      0.01                      Failure data
       1.0      100.0      1.0            Repair data
           system    power        1       Input signal
           system    command      1       Input signal
           PUMP1     process      1       Input signal
           PUMP2     process      1       Input signal
           PUMP3     process      1       Input signal
           system    process      1       Output signal
standby                                   Initial system state
         1                                System success criteria
         0                                Number of external events
```

```
SIMULATION OF GO-FLOW LIGHT BULB PROBLEM
     20.00                                                   Time of simulation
         0                                                   Type of run (0 for normal)
      1000                                                   Number of trials
         7                                                   Number of time points
         0                                                   Type of time distribution
      0.00
      1.00
      9.99
     10.00                                                   Time points
     11.00
     15.00
     20.00
         5                                                   Number of components
BATTERY    passive    standby       1     2                  Component number one
      0.1        0.0                                         Failure data
      1.0        1.0        0.0                              Repair data
           system     process       0                        Input signal
           SWITCH1    process       0                        Output signal
           SWITCH2    process       0                        Output signal
SWITCH1    switch     open          3     1                  Component number two
      0.3        0.0                                         Failure data
      1.0        1.0        0.0                              Repair data
           system     command       0                        Input signal
           system     power         1                        Input signal
           BATTERY    process       0                        Input signal
           LIGHT1     process       0                        Output signal
SWITCH2    switch     open          3     1                  Component number three
      0.3        0.0                                         Failure data
      1.0        1.0        0.0                              Repair data
           system     command       0                        Input signal
           system     power         1                        Input signal
           BATTERY    process       0                        Input signal
           LIGHT2     process       0                        Output signal
LIGHT1     passive    standby       1     1                  Component number four
      0.2      0.001                                         Failure data
      1.0        1.0        0.0                              Repair data
           SWITCH1    process       0                        Input signal
           system     process       0                        Output signal
LIGHT2     passive    standby       1     1                  Component number five
      0.2      0.001                                         Failure data
      1.0        1.0        0.0                              Repair data
           SWITCH2    process       0                        Input signal
           system     process       0                        Output signal
standby                                                      Initial system state
         1                                                   System success criteria
         3                                                   Number of external events
      0.00          0                                        External event #1, Time, #Comps.
         1                                                   Number signals
system     BATTERY    process                                Signal
         1                                                   New strength
      0.00          0                                        External event #2, Time, #Comps.
         1                                                   Number signals
system     SWITCH1    command                                Signal
        -1                                                   New strength
     10.00          0                                        External event #3, Time, #Comps.
         1                                                   Number signals
system     SWITCH2    command                                Signal
        -1                                                   New strength
```

```
TEST OF THE TANK PORTION OF THE PROGRAM
   1000.00
         0
      1000
       201
         1
         3
unit1     valve     open          3    1
       0.0   0.00312
       1.0        1.0        0.0
          system    power         1
          tank      process       1
          tank      command       1
          nowhere   process       1
unit2     valve     open          3    1
       0.0   0.00456
       1.0        1.0        0.0
          system    power         1
          system    process       1
          tank      command       1
          tank      process       1
unit3     valve     closed        3    1
       0.0    0.0057
       1.0        1.0        0.0
          system    power         1
          system    process       1
          tank      command      -1
          tank      process       0
   standby
         1
         0
```

Appendix E

Sample Output Files

```
SINGLE COMPONENT, EXP REPAIR AND FAILURE, DUAL REPAIR STATES
  10000.00
         0
       100
        21
         1
         1
COMPONENT passive   operating      1    1
   0.          .01000
   1.00000 100.00000   1.00000
          system    process        1
          system    process        1
standby
         1
         0
```

AFTER 100 TRIALS
AND
OVER A TIME PERIOD OF 10000 HOURS
THE AVERAGE SYSTEM UNAVAILABILITY IS AS FOLLOWS

The minimum is:	.5510
The 1st percentile is:	.5510
The 5th percentile is:	.5804
The 25th percentile is:	.6343
The 40th percentile is:	.6538
The 50th percentile is:	.6618
The 60th percentile is:	.6740
The 75th percentile is:	.7002
The 95th percentile is:	.7440
The 99th percentile is:	.7579
The maximum is:	.7732
The mean is:	.6644
The variance is:	.0023

AFTER 100 TRIALS

THE TIME DEPENDENT UNAVAILABILITY IS AS FOLLOWS

TIME	UNAVAILABILITY
0.	0.
500.00	.6300
1000.00	.7000
1500.00	.7000
2000.00	.6800
2500.00	.6700
3000.00	.6500
3500.00	.6700
4000.00	.7200
4500.00	.6900
5000.00	.6400
5500.00	.5900
6000.00	.6300
6500.00	.6800
7000.00	.6500
7500.00	.6800
8000.00	.6900
8500.00	.6800
9000.00	.6100
9500.00	.7000
10000.00	.6400

point	1	is	.5510
point	2	is	.5622
point	3	is	.5700
point	4	is	.5787
point	5	is	.5804
point	6	is	.5836
point	7	is	.5883
point	8	is	.5956
point	9	is	.5962
point	10	is	.5967
point	11	is	.5976
point	12	is	.5997
point	13	is	.6006
point	14	is	.6056
point	15	is	.6095
point	16	is	.6121
point	17	is	.6122
point	18	is	.6167
point	19	is	.6179
point	20	is	.6223
point	21	is	.6233
point	22	is	.6264
point	23	is	.6321
point	24	is	.6342
point	25	is	.6343
point	26	is	.6371
point	27	is	.6374
point	28	is	.6399
point	29	is	.6414
point	30	is	.6430
point	31	is	.6444
point	32	is	.6454
point	33	is	.6464
point	34	is	.6465
point	35	is	.6477
point	36	is	.6481
point	37	is	.6494
point	38	is	.6500
point	39	is	.6525
point	40	is	.6538
point	41	is	.6540
point	42	is	.6544
point	43	is	.6547
point	44	is	.6555
point	45	is	.6568
point	46	is	.6586
point	47	is	.6597
point	48	is	.6617
point	49	is	.6617
point	50	is	.6618
point	51	is	.6620
point	52	is	.6626
point	53	is	.6633

point	54	is	.6647
point	55	is	.6661
point	56	is	.6671
point	57	is	.6674
point	58	is	.6680
point	59	is	.6694
point	60	is	.6740
point	61	is	.6742
point	62	is	.6756
point	63	is	.6763
point	64	is	.6821
point	65	is	.6835
point	66	is	.6850
point	67	is	.6875
point	68	is	.6876
point	69	is	.6879
point	70	is	.6922
point	71	is	.6932
point	72	is	.6967
point	73	is	.6978
point	74	is	.6996
point	75	is	.7002
point	76	is	.7023
point	77	is	.7040
point	78	is	.7049
point	79	is	.7064
point	80	is	.7064
point	81	is	.7084
point	82	is	.7085
point	83	is	.7097
point	84	is	.7136
point	85	is	.7146
point	86	is	.7180
point	87	is	.7185
point	88	is	.7218
point	89	is	.7243
point	90	is	.7248
point	91	is	.7260
point	92	is	.7273
point	93	is	.7375
point	94	is	.7416
point	95	is	.7440
point	96	is	.7502
point	97	is	.7523
point	98	is	.7541
point	99	is	.7579
point	100	is	.7732

```
SIMULATION OF GO-FLOW LIGHT BULB PROBLEM
      20.00
          0
       1000
          7
          0
       0.
       1.00
       9.99
      10.00
      11.00
      15.00
      20.00
          5
 BATTERY    passive   standby        1     2
     .10000   0.
   1.00000   1.00000   0.
            system    process        0
            SWITCH1   process        0
            SWITCH2   process        0
 SWITCH1    switch    open           3     1
     .30000   0.
   1.00000   1.00000   0.
            system    command        0
            system    power          1
            BATTERY   process        0
            LIGHT1    process        0
 SWITCH2    switch    open           3     1
     .30000   0.
   1.00000   1.00000   0.
            system    command        0
            system    power          1
            BATTERY   process        0
            LIGHT2    process        0
 LIGHT1     passive   standby        1     1
     .20000    .00100
   1.00000   1.00000   0.
            SWITCH1   process        0
            system    process        0
 LIGHT2     passive   standby        1     1
     .20000    .00100
   1.00000   1.00000   0.
            SWITCH2   process        0
            system    process        0
 standby
          1
          3
       0.          0
          1
 system     BATTERY   process
          1
       0.          0
          1
```

```
system     SWITCH1    command
        -1
     10.00          0
         1
system     SWITCH2    command
        -1
```

AFTER 1000 TRIALS

THE TIME DEPENDENT UNAVAILABILITY IS AS FOLLOWS

TIME	UNAVAILABILITY
0.	.5090
1.00	.5090
9.99	.5120
10.00	.2940
11.00	.2940
15.00	.2990
20.00	.3020

AFTER 1000 TRIALS
AND
OVER A TIME PERIOD OF 20 HOURS
THE AVERAGE SYSTEM UNAVAILABILITY IS AS FOLLOWS

The minimum is:	.0000
The 1st percentile is:	.0000
The 5th percentile is:	.0000
The 25th percentile is:	.0000
The 40th percentile is:	.0000
The 50th percentile is:	.0044
The 60th percentile is:	.0492
The 75th percentile is:	1.0000
The 95th percentile is:	1.0000
The 99th percentile is:	1.0000
The maximum is:	1.0000
The mean is:	.3416
The variance is:	.2024

```
SIMULATION OF GO-FLOW LIGHT BULB PROBLEM
     20.00
         0
     10000
         7
         0
       0.
      1.00
      9.99
     10.00
     11.00
     15.00
     20.00
         5
BATTERY    passive    standby        1      2
  .10000    0.
  1.00000   1.00000    0.
           system     process        0
           SWITCH1    process        0
           SWITCH2    process        0
SWITCH1    switch     open           3      1
  .30000    0.
  1.00000   1.00000    0.
           system     command        0
           system     power          1
           BATTERY    process        0
           LIGHT1     process        0
SWITCH2    switch     open           3      1
  .30000    0.
  1.00000   1.00000    0.
           system     command        0
           system     power          1
           BATTERY    process        0
           LIGHT2     process        0
LIGHT1     passive    standby        1      1
  .20000     .00100
  1.00000   1.00000    0.
           SWITCH1    process        0
           system     process        0
LIGHT2     passive    standby        1      1
  .20000     .00100
  1.00000   1.00000    0.
           SWITCH2    process        0
           system     process        0
standby
         1
         3
      0.            0
         1
system     BATTERY    process
         1
      0.            0
         1
```

```
system     SWITCH1   command
       -1
     10.00          0
        1
system     SWITCH2   command
       -1
```

AFTER10000 TRIALS

THE TIME DEPENDENT UNAVAILABILITY IS AS FOLLOWS

TIME	UNAVAILABILITY
0.	.4993
1.00	.4999
9.99	.5052
10.00	.2757
11.00	.2763
15.00	.2787
20.00	.2814

AFTER10000 TRIALS
AND
OVER A TIME PERIOD OF 20 HOURS
THE AVERAGE SYSTEM UNAVAILABILITY IS AS FOLLOWS

The minimum is:	.0000
The 1st percentile is:	.0000
The 5th percentile is:	.0000
The 25th percentile is:	.0000
The 40th percentile is:	.0000
The 50th percentile is:	.0033
The 60th percentile is:	.0289
The 75th percentile is:	1.0000
The 95th percentile is:	1.0000
The 99th percentile is:	1.0000
The maximum is:	1.0000
The mean is:	.3225
The variance is:	.1944

```
TEST OF THE TANK PORTION OF THE PROGRAM
   1000.00
         0
      1000
       201
         1
         3
unit1      valve      open          3    1
       0.0   0.00312
       1.0        1.0        0.0
           system     power         1
           tank       process       1
           tank       command       1
           nowhere    process       1
unit2      valve      open          3    1
       0.0   0.00456
       1.0        1.0        0.0
           system     power         1
           system     process       1
           tank       command       1
           tank       process       1
unit3      valve      closed        3    1
       0.0    0.0057
       1.0        1.0        0.0
           system     power         1
           system     process       1
           tank       command      -1
           tank       process       0
   standby
         1
         0
```

AFTER 1000 TRIALS

THE TIME DEPENDENT UNAVAILABILITY ANALYSIS IS AS FOLLOWS

TIME	UNAVAILABILITY
0.	0.
5.00	0.
10.00	0.
15.00	0.
20.00	0.
25.00	.0010
30.00	.0010
35.00	.0040
40.00	.0060
45.00	.0090
50.00	.0120
55.00	.0130
60.00	.0140
65.00	.0160
70.00	.0170
75.00	.0200
80.00	.0230
85.00	.0240
90.00	.0270
95.00	.0300
100.00	.0320
105.00	.0330
110.00	.0370
115.00	.0400
120.00	.0430
125.00	.0460
130.00	.0510
135.00	.0580
140.00	.0640
145.00	.0690
150.00	.0720
155.00	.0740
160.00	.0740
165.00	.0760
170.00	.0780
175.00	.0830
180.00	.0870
185.00	.0870
190.00	.0880
195.00	.0920
200.00	.0940
205.00	.0950
210.00	.1010
215.00	.1020
220.00	.1120
225.00	.1160
230.00	.1180
235.00	.1210
240.00	.1230
245.00	.1260
250.00	.1300

255.00	.1350
260.00	.1390
265.00	.1430
270.00	.1440
275.00	.1500
280.00	.1560
285.00	.1570
290.00	.1590
295.00	.1630
300.00	.1650
305.00	.1670
310.00	.1730
315.00	.1770
320.00	.1790
325.00	.1800
330.00	.1810
335.00	.1830
340.00	.1850
345.00	.1850
350.00	.1860
355.00	.1890
360.00	.1900
365.00	.1920
370.00	.1940
375.00	.1970
380.00	.2010
385.00	.2020
390.00	.2070
395.00	.2100
400.00	.2100
405.00	.2100
410.00	.2120
415.00	.2140
420.00	.2150
425.00	.2170
430.00	.2190
435.00	.2210
440.00	.2220
445.00	.2240
450.00	.2280
455.00	.2290
460.00	.2300
465.00	.2340
470.00	.2370
475.00	.2370
480.00	.2370
485.00	.2400
490.00	.2420
495.00	.2420
500.00	.2430
505.00	.2470
510.00	.2480
515.00	.2500
520.00	.2560
525.00	.2570

530.00	.2580
535.00	.2600
540.00	.2630
545.00	.2630
550.00	.2640
555.00	.2640
560.00	.2660
565.00	.2670
570.00	.2700
575.00	.2720
580.00	.2740
585.00	.2750
590.00	.2780
595.00	.2790
600.00	.2800
605.00	.2800
610.00	.2810
615.00	.2820
620.00	.2840
625.00	.2850
630.00	.2860
635.00	.2880
640.00	.2890
645.00	.2900
650.00	.2920
655.00	.2930
660.00	.2940
665.00	.2940
670.00	.2950
675.00	.2970
680.00	.2990
685.00	.3010
690.00	.3020
695.00	.3060
700.00	.3070
705.00	.3090
710.00	.3090
715.00	.3090
720.00	.3090
725.00	.3100
730.00	.3100
735.00	.3110
740.00	.3130
745.00	.3160
750.00	.3170
755.00	.3180
760.00	.3180
765.00	.3200
770.00	.3210
775.00	.3210
780.00	.3210
785.00	.3210
790.00	.3220
795.00	.3220
800.00	.3220

805.00	.3230
810.00	.3240
815.00	.3240
820.00	.3250
825.00	.3250
830.00	.3250
835.00	.3250
840.00	.3260
845.00	.3260
850.00	.3260
855.00	.3260
860.00	.3260
865.00	.3260
870.00	.3270
875.00	.3270
880.00	.3270
885.00	.3270
890.00	.3270
895.00	.3290
900.00	.3300
905.00	.3310
910.00	.3320
915.00	.3330
920.00	.3330
925.00	.3340
930.00	.3340
935.00	.3350
940.00	.3350
945.00	.3350
950.00	.3350
955.00	.3360
960.00	.3360
965.00	.3360
970.00	.3360
975.00	.3360
980.00	.3360
985.00	.3370
990.00	.3370
995.00	.3380
1000.00	.3380

AFTER 1000 TRIALS

THE UNAVAILABILITY DISTRIBUTION DATA IS AS FOLLOWS

The minimum is:	.0000
The 1st percentile is:	.0000
The 5th percentile is:	.0000
The 25th percentile is:	.0000
The 40th percentile is:	.0000
The median is:	.0000
The mean is:	.2155
The 60th percentile is:	.0000
The 75th percentile is:	.4840
The 95th percentile is:	.8701
The 99th percentile is:	.9540
The maximum is:	.9790
The variance is:	.1085