A TAXONOMICAL REVIEW OF SOFTWARE VERIFICATION TECHNIQUES: AN ILLUSTRATION USING DISCRETE-EVENT SIMULATION

by

Richard B. Whitner

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

Osman Balci, Chairman

Richard E. Nance

Jeffrey D. Tew

October, 1988

Blacksburg, Virginia

A TAXONOMICAL REVIEW OF SOFTWARE VERIFICATION TECHNIQUES: AN ILLUSTRATION USING DISCRETE-EVENT SIMULATION

by

Richard B. Whitner

Osman Balci, Chairman

Computer Science and Applications

(ABSTRACT)

The use of simulation and modeling as a technique for solving today's complex problems is ever-increasing. Correspondingly, the demands placed on the software which serves as a computer-executable representation of the simulation model are increasing. With the increased complexity of simulation models comes greater need for model verification, particularly programmed model verification. Unfortunately, current model verification technology is lacking in techniques which satisfy the verification needs. Specifically, there are few guidelines for performing programmed model verification. There is, however, an abundance of software verification techniques which are applicable for simulation model verification. An extensive review of techniques applicable for simulation programmed model verification is presented using the simulation and modeling terminology. A taxonomy for programmed model verification methods is developed. The usefulness of the taxonomy is twofold: (1) the taxonomy provides an approach for applying software verification techniques to the problem of programmed model verification, and (2) the breadth of the taxonomy provides a broad spectrum of perspectives from which to assess the credibility of simulation results. A simulation case study demonstrates the usefulness of the taxonomy and some of the verification techniques.

By referencing this work, one can determine what, and when, techniques can be used throughout the development life cycle. He will know how to perform each technique, how difficult each will be to apply, and how effective the technique will be. The simulation

modeler—as well as the software engineer—will find the taxonomy and techniques valuable tools for guiding verification efforts.

Acknowledgements

My sincere appreciation is given to my thesis advisor, Dr. Osman Balci, for his guidance and insight into the intricacies of simulation and modeling and the needs for improved credibility assessment of simulation models. His preciseness and attention to detail has made it much easier to maintain the focus of this research. I must also thank Drs. James D. Arthur and Richard E. Nance for stimulating my interest in the realm of software quality assurance. Thanks also goes to Dr. Nance and Dr. Jeffrey D. Tew for serving on my thesis advisory committee and offering me their critical opinions.

I am indebted to and for their help in preparing some of the figures and other technical aspects of the manuscript, and to numerous other individuals who offered ideas, suggestions, prayers, and words of encouragement.

Most of all, I express my deepest gratitude to my wife, , and my son, , for their unending patience and support which sustained me during this adventure. I must not forget the ultimate source of my strength, my Lord Jesus Christ:

"I can do all things through Him who strengthens me." Phillipians 4:13 (NAS Bible)

Table of Contents

List of Illustrations

List of Tables

1.0 INTRODUCTION

Software verification is a major concern of today's software engineering community. It is a well known fact among software developers that over 50 percent of the development effort and resources go into the verification process. This process encompasses the entire software development life cycle from inception to implementation. It is no wonder that a tremendous amount of research has been devoted to the area of software verification.

In the area of simulation, verification is also a crucial element. The simulation model life cycle has a much broader scope than does the general software life cycle, and in fact includes a subordinate cycle which is analogous to that for software (the development of the simulation programmed model). The entire simulation model life cycle is monitored by a series of credibility assessment stages [Balci 1987]. These credibility assessment stages (CASs) seek to assure the acceptability of the simulation results.

The ultimate goal of a simulation study is to produce a *sufficiently correct* problem solution that will be *accepted* and *used* by the decision maker [Balci and Nance 1985]. The entire fate of a simulation study (and even the modeler!) hangs in the balance of successful model verification. If the modeler cannot convince the project sponsor that the simulation study is suf-

ficiently correct, the model results will not be accepted. Likewise, if the modeler convinces the sponsor to accept the results when in fact they are not sufficiently credible, the longterm result will inevitably be disastrous and the modeler's reputation will suffer. The modeler stands to gain through a sound model verification program. It is unfortunate that with so much at stake, so little attention has been given to certain key aspects of the simulation model verification process.

Simulation experts have focused much attention on *validating* simulation models, i.e., assuring that the model is sufficiently representative of the system under study. For example, the problem definition is carefully examined for completeness and accuracy. The problem domain is tightly bounded and carefully compared with the real system to ensure that the model completely encapsulates the real system. The programmed model inputs and outputs are checked for validity, and then the model is experimented with and the results of the experimentation validated. Along the way, however, the importance of the programmed model development process and its accompanying verification gets overlooked. In a survey of leading textbooks on discrete-event simulation methodology published since 1970 [Balci 1987], it was found that only 0.3% of the combined pages of these books treat programmed model verification. Verification of the programmed model is simply viewed as program debugging [Pace 1987]. The motivation to retain the programmed model after the study is small; the model is often discarded.

This perfunctory view of the programmed model has caused many simulation experts to overlook the area of programmed model verification. Quite often, neither sufficient time nor resources are allocated for it. As one simulation study from the aerospace industry admitted, programmed model verification has not been thorough, "due primarily to the twin constraints of cost and schedule." [Innis et al. 1977] As simulation models continue to grow in size and complexity, the simulation community is beginning to recognize the dire need for engineering

quality models. This awareness has been brought about in large part by the need to retain and maintain the programmed models used for simulation study for extended periods of time.

Perhaps one of the factors resulting in the lack of programmed model verification technology is the terminology gap between the software engineering and simulation communities. For example, in software engineering terminology a *software engineer* develops *programs*. In simulation, a *modeler* builds *models*. From a software engineering viewpoint, the person who designs the program is termed an *analyst*, a *developer*, or an *engineer*, and the person responsible for coding the program is a *programmer*. The simulation counterpart of all of these is the *modeler*. As opposed to developing a *product*, the modeler conducts a *study*. These are just a few of the differences between the two "languages." It is only natural to expect differences in terminology and lack of communication given that the majority of simulation modelers have backgrounds in statistics, industrial engineering and operations research, and related fields—not software engineering. The textbook survey mentioned previously exemplifies this gap. A literature review conducted by Balci and Sargent [1984] further supports the existence of such a gap. The Life Cycle of a Simulation Study shown in Figure 1 (reprinted from [Balci 1987]) illustrates such differences.

All software verification techniques are applicable to programmed model verification. Only the usefulness and practicality of the techniques may vary between the two domains. Some techniques which are not considered practical software engineering verification alternatives serve model verification very well. Other techniques serve both communities equally well. This thesis is intended to reduce the communication gap between the software engineering and simulation communities by presenting software verification techniques and showing their applicability to programmed model verification. Because the emphasis of this thesis is on verifying simulation models, the terminology used is slanted towards the field of simulation. The software engineer will still, no doubt, find the taxonomy and presentation of techniques beneficial to him as well.

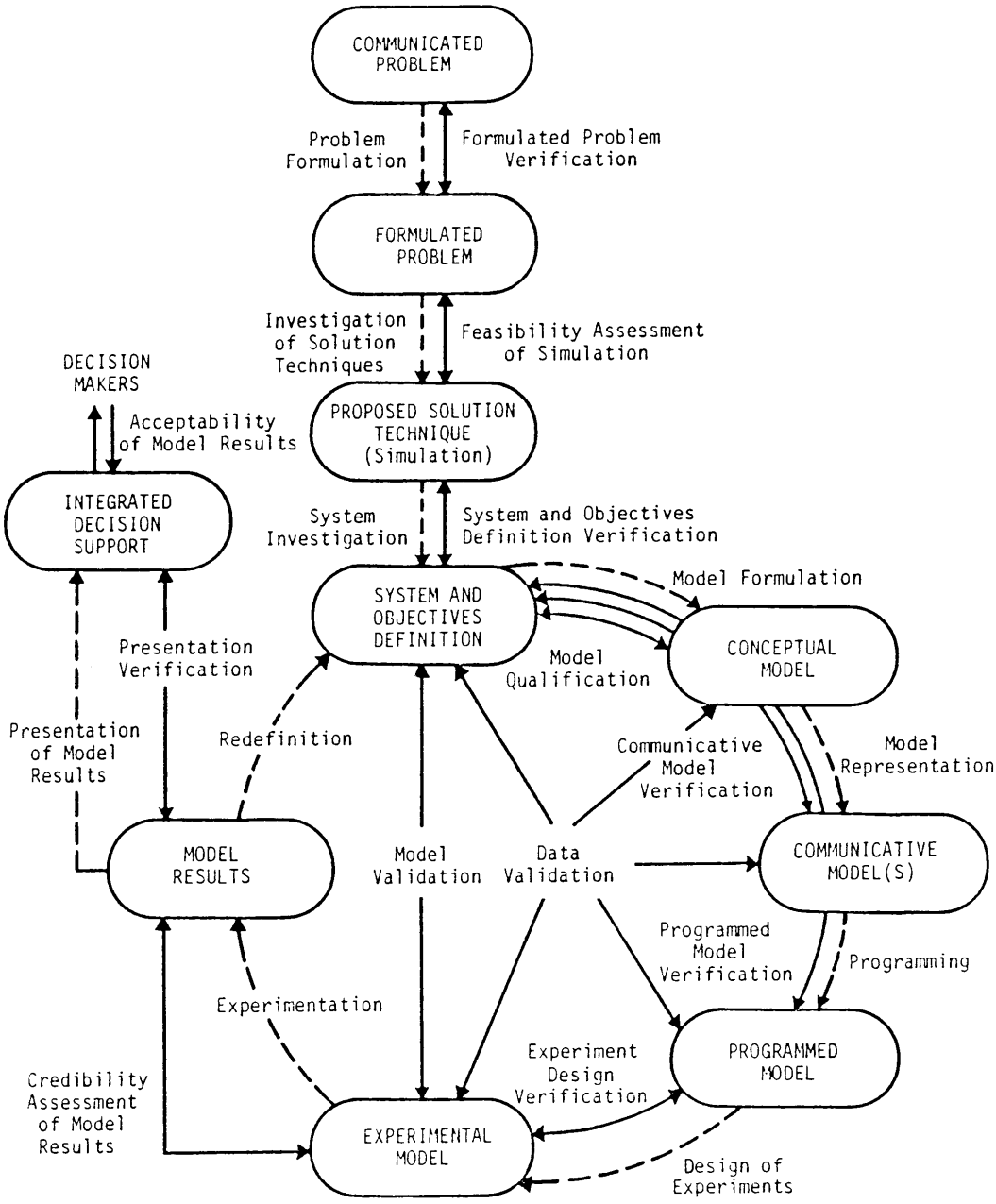


Figure 1. The Life Cycle of a Simulation Study (reprinted from [Balci 1987])

The remainder of this chapter provides discussion of the general concepts of verification, validation, and model credibility, background discussion of the organization aspects of the testing process, and an overview of the research approach taken for this work. Chapter 2 presents the Simulation Model Development Life Cycle. A taxonomy for PMV methods is developed in Chapter 3 along with a simulation case study to illustrate the usefulness of the taxonomy. Chapter 4 expounds on the methods presented in the taxonomy. Chapter 5 provides concluding remarks.

1.1 Verification, Validation, and Model Credibility

All simulation models are descriptive models, i.e., they make no value judgment on the "goodness" or "badness" of model results. A simulation model is neither absolutely right nor wrong. Rather, the model is assessed with respect to its intended study objectives. Credibility, quality, validity, and verity are judged with respect to the study objectives. [Balci 1987]

There is much confusion concerning the difference between verification and validation. *Validation* involves comparing the simulation model with the real system to determine the accuracy of the model representation. Whenever a model or model component is *compared* with reality, validation is being performed. *Verification*, on the other hand, is concerned with the accuracy, completeness, and unambiguity of the *transformation* of a model from one form into another during its development life cycle. As one expert once put it, validation deals with building the right model, verification deals with building the model right [Boehm 1984]. PMV is substantiation that the executable simulation model (i.e., the programmed model) is a sufficiently accurate transformation of the model specification. The implies ascertaining that the programmed model is free of transformation errors.

A recent Department of Defense (DoD) simulation study [GAO/PEMD-88-3, 1987] named "providing evidence of a verification effort" as one of the leading factors contributing to the credibility of simulation results. Not surprisingly, the study found verification efforts to be minimal. What was surprising, and even alarming, was the fact that in two of three case study simulations of systems costing over $200 million, *no* documentary evidence of verification was available. The third had no specific verification efforts identified. The study states:

> "The lack we found of documented evidence of [programmed model] verification presents a clear threat to the credibility of the three simulations."

A model which is not sufficiently credible, regardless of how impressive or elegant it might be, serves absolutely no purpose. PMV is one means of assessing the credibility of the model.

Three types of errors committed during a simulation study are identified in the literature. These are summarized in [Balci 1987] and presented here. *Type I Error* is committed when a sufficiently credible model is rejected by the decision makers. The probability of this type of error occurring is called *model builder's risk* [Balci and Sargent 1981]. Reasons for type I error occurring range from failure on the part of the modeler to convince study sponsors of the model's credibility, to non-technical ("political") factors. Model appearance and the manner in which model results are presented may be the difference between success and failure.

Type II Error is committed when an insufficiently credible model is accepted by the study sponsors. The likelihood of this error occurring is known as *model user's risk* [Balci and Sargent 1981]. Type II error is a particularly dangerous type of error from two perspectives. First, the study results are invalid—they have no meaning. Any decisions made on the basis of the study are unfounded. Making a decision on the basis of misinformation is quite often more dangerous than making one with no information at all. Given the mission-critical nature of many of today's systems—systems which are implemented on the basis of simulation study results—the occurrence of type II error can have grave consequences. The other impact of

type II error is on the reputation of the modeler. Inevitably, a system based on an invalid model is going to fail. The source of the failure will eventually be traced back to the simulation study.

Type III Error occurs when the formulated problem does not completely contain the ***real*** problem [Balci and Nance 1985]. This corresponds to solving the wrong problem. The occurrence of Type III Error results in either Type II Error or unsuccessful ending of the simulation study.

Balci [1987] presents a hierarchy of credibility assessment stages for evaluating the acceptability of simulation (model) results. Following this scheme will help the modeler reduce the likelihood that the three types of errors will occur.

1.2 Background

This section provides background discussion of model instrumentation, test preparation, test data generation, and mutation testing. The discussion given here is preliminary to more extensive coverage of testing given in Chapter 4. The sections here are intended to provide an organizational flavor of the testing process, as opposed to the more functional discussion of testing techniques given later.

1.2.1 Instrumentation

In this section, instrumentation is introduced. Instrumentation lays the foundation for extracting execution information from the model. In Chapters 3 and 4, instrumentation is referred to

when discussing various verification techniques. Therefore it is necessary that the reader be familiar with instrumentation.

Instrumentation is a process of inserting *trap codes* (also called *hooks, breakpoints,* and *probes*) into the programmed model for the purpose of extracting information during the model's execution. This additional code monitors the activity of the model, and, when events of interest occur, "traps" (interrupts or freezes) the model's execution and records the interesting data. When the data has been recorded, control is passed back to the model, which then continues its execution as if uninterrupted.

One method of instrumenting the model is to add test submodels (probes) to the existing code which gain control of the model when execution begins, and then turns control over to the programmed model. The added code acts as a "shell" around the model. The test submodels themselves are either added to, and compiled with, the source code, or simply linked with the programmed model object modules. "Normal" execution proceeds under the auspices of these added submodels. As dictated by the test submodels, execution is *trapped* (interrupted) and the model state is dumped. An alternative approach is not to use an execution shell but simply let the trap code respond to signals generated during execution, such as a clock signal or other hardware interrupt. These types of instrumentation generally require less knowledge of the programmed model structure than other instrumentation methods, but are limited in the information that they produce, and may require considerable effort to analyze the results. Gathering more detailed information requires a different type, and more judicial placement, of probes.

Another method of instrumentation, one which is especially powerful for PMV, is the placement of *assertion statements* at key points in the model's control structure. The assertions are reflections of model assumptions. An assertion statement can be constructed to compare

actual programmed model state with the underlying assumptions to assure that the assumptions are not being violated, i.e., the model is behaving within its bounds.

Another means of instrumenting the model is to add instructions which use the programmed model symbol table and map (created during model source code compilation) to relate the executable code to the source code. In doing this, the execution history can be traced and the model profiled in greater detail. Events which take place during the model's execution can be traced back to their origin in the source code. Counts of the number of times a source statement was executed or a variable referenced are easily obtained.

More in-depth instrumentation involves analyzing the model data flow and control flow structures and identifying optimal locations to place probes and assertions. Things to look for in probe placement include the critical nature of the various model structures, the effectiveness of the location (the amount of data gathered per probe), and the runtime cost of the probe. A good probe location would be one which validates the activities of the model at key points on its control path and collects the maximum amount of state data, without seriously altering the performance of the model. Poor probe placement would be a location which provided a proper subset of information available at another location, or which degraded performance beyond tolerable limits. Symbolic analysis techniques are useful in determining optimal probe placement.

Digressing for just a moment, the problem of performance degradation represents a fundamental issue in using instrumented code to verify software, especially real-time or tightly synchronized software. Simulation software faces the same difficulties as production level software in this regard: instrumentation degrades the performance of the software. This is, however, a key point of variation in the verification needs of simulation versus non-simulation software. The bottom-line of PMV is to realize a *sufficiently valid* model—not just one that is correct—for the purpose of experimenting with the model. It is possible for a simulation pro-

grammed model to be correct (according to its specifications) but not operating validly within the constraints of the model; hence an invalid specification. Performance must be considered a fair trade-off for insuring the correctness, and ultimately the validity, of the model. This subject will be addressed further in another section.

A useful method of probe placement, in addition to structural analysis, is to profile the model using one instrumentation technique, then use the collected data to locate areas in greater need of instrumentation. This is a progressive approach to instrumentation. If the collected data is maintained on-line, much of this process can be automated.

To facilitate testing, areas of instrumented code can often be made active or inactive through the use of compiler directives. This allows testing of selected portions of the model. In the informal sense, the instrumented code is simply commented out by the modeler. In a more formal sense, instrumentation features are provided as an extension of the language. When testing is complete, the probes can be turned off altogether. The probe information provides an excellent source of documentation. Leaving assertions in the model during experimentation can help validate the model. Still another way to control the testing process is through automated testing environments (testbeds) which are designed specifically for testing purposes.

1.2.2 Test Preparation

As part of the simulation model life cycle, a comprehensive test plan should be formed. This plan should be formed in the earliest stages of the life cycle. As the first specifications are being prepared, they should be analyzed to extract test data information. The importance of this early test preparation is illustrated by the recently awarded $508 million, five-year DoD contract to develop a testbed for testing its Strategic Defense Initiative (SDI) system.

A model testbed provides an environment for performing programmed model testing. This means that the testbed supplies test data, provides a mechanism to exercise the model with the test data, and collects the test results. Using the data available through static analysis of the code *and* specifications, the types of data needed, the interfaces required, the areas in need of testing, optimal probe location, etc., can be determined to construct a sound testbed. This information, coupled with test data from symbolic test data generation techniques, can be used to produce a very comprehensive and efficient test plan. Pressman [1987] and Grady and Caswell [1987] provide excellent discussion of preparing, managing, and measuring the testing process.

1.2.3 Test Data Generation

The challenge in test data generation is to produce inputs that will force execution to follow selected model paths. Not only should paths be traversed, *interesting* data about model execution along the paths should be revealed. Among the interesting situations are those which occur along the boundaries of various model input subdomains, such as the maximum and minimum values of the subdomain, or those which have been randomly selected for the input domain.

As will be discussed later, symbolic execution yields a symbolic execution tree depicting the entire model path structure. This tree is, in fact, a set of equality and inequality constraints on the input variables of the model [Ramamoorthy et al. 1976]. These constraints define a subset of the input data space which will cause execution of selected model paths. Test data is generated from these subsets. If it can be determined that the set of constraints is inconsistent, then the given path is non-executable [Clarke 1976]. Hence test data need not be selected for that path and the path can be struck from further consideration. Other test data

generation techniques [Myers 1979; Chusho 1987; Beizer 1983] produce high-yield test cases which reduce testing needs by testing *interesting* situations along critical model paths.

Once test data has been generated, the expected results of the individual test cases must be determined. Again, symbolic execution provides a solution. Since the leaves of the symbolic execution tree are labeled as expressions given in terms of model inputs, test data values may be substituted and the expression evaluated. Unfortunately, this is not always as straightforward as it sounds, with the difficulties in performing symbolic execution becoming a major problem. The overall test data generation problem is further complicated when the constraints turn out to be non-linear. This gives rise to the need for non-linear programming techniques to solve the constraint expressions, even to the point of employing systematic trial and error [Ramamoorthy et al. 1976].

1.2.4 Mutation Testing

Mutation testing [Budd et al. 1980; DeMillo et al. 1978] is a technique which aids in evaluating the effectiveness of the testing process. It involves seeding the model with known errors, or mutations, and then re-testing the mutated model.

The basis for mutation testing is simple. Given a "reasonably correct" model (i.e., testing reveals few or no errors), seed the model with known errors, retest the model, and see if all of the seeded errors were found. Further, for the errors found, see how meaningful the error results produced are. If the test process fails to reveal all mutants, it is an indication that the test coverage is not adequate. If the test results do little to help locate a mutant, then the test results are not meaningful.

Mutation testing is expensive in the time required to effectively seed the model. It is also expensive in executing the model to obtain the results. Clearly, however, it is an effective way of evaluating test effectiveness.

1.3 Research Approach

The goal of this thesis is to provide the modeler with a battery of usable PMV techniques. A shortage of PMV techniques stems from the lack of effective communication between the software engineering and simulation communities. This communication gap is due to differences in terminology (as a consequence of different backgrounds) and the lack of a comprehensive, simulation-oriented picture of the verification process. It has already been pointed out that there are ample verification techniques available within the software engineering community. However, these techniques are neither simulation-oriented nor comprehensively portrayed. To attain the stated objective, these two challenges must be met.

The first step is to adopt a simulation model life cycle. All studies follow a development life cycle. Somewhere there is a beginning to the study, somewhere there is an end, and somewhere in between the beginning and end there is development. If the life cycle is not well defined, problems are inevitable. Balci [1987] and Nance [1981] present a complete life cycle of a simulation study which provides the proper framework for conducting simulation studies.

The next step is to develop a taxonomy of software verification techniques applicable for simulation programmed model verification. The verification techniques are amalgamated according to their basis for justifying the accuracy of the model translation. Each category will be examined for important characteristics such as level of mathematical formalism, complexity of use, cost, effectiveness, etc. Advantages and disadvantages of each category will

be identified, and the methods within each category will be related to the CASs in which they might be applied. The usefulness of the taxonomy for PMV will be discussed.

Following the development of the taxonomy, a simulation case study will be conducted. The case study will draw techniques from the taxonomy and apply the techniques to the problem of verifying portions of a simulation programmed model. The usefulness of the taxonomy in guiding the verification will be evident.

Finally, each category will be expounded with respect to the taxonomy developed. The characteristics, advantages, and disadvantages of each will be discussed, along with a description of each verification technique within the category. The discussion will be aided with meaningful examples from the case study.

2.0 SIMULATION MODEL DEVELOPMENT

We begin this chapter with a brief scenario of a simulation model development.

An organization recognizes a problem that needs to be solved. Someone mentions computer simulation as a possible means of solving the problem. An "expert" is obtained to perform a computer simulation of the problem. The organization with the problem is the study sponsor. The "expert" is the modeler. The sponsor describes its view of the problem. The modeler takes this view, refines it to make it workable, and writes a computer program to simulate the problem. When the program is finished, the modeler runs the programmed "model" and detects what he thinks is the source of the problem. The modeler experiments a bit with the model's parameters in an effort to solve the problem, but while doing so, detects another problem. After numerous iterations of this experimentation and problem detection, it becomes apparent that something has gone amiss. Time runs out on the project. The modeler must save face, so he exerts much effort in selling the virtues of the model. The sponsor either realizes the model for what it is and rejects it, or falls for the modeler's sales pitch. In either case, the study has been a failure.

The model in this scenario was doomed from the start. In the first place, a problem never originates as a "simulation problem." Computer simulation is merely an alternative solution technique in an approach to solving a problem. The second major point of departure was in developing a solution to an as-yet determined problem. The other problems with the study unfold as the development cycle continues.

The point here is that any problem solving process, be it simulation or otherwise, follows a development life cycle. The question to be answered is, "Are there well-formulated guidelines for solving the problem?" In the above scenario, the answer is a resounding "NO!" Unfortunately, too many simulation studies follow the same path. In this chapter, the simulation model development life cycle (SMDLC) is presented. The purpose of the life cycle is to serve as a framework to guide project development.

2.1 The Simulation Model Development Life Cycle

As mentioned above, a simulation study does not originate as a simulation study. Rather, simulation is a technique selected to solve a given problem. Balci [1987] and Nance [1981] present a complete life cycle of a simulation study, of which the SMDLC is a subset. The life cycle is shown in Figure 1. It provides a framework to guide the modeler through the simulation study. The ovals represent the phases of the life cycle. The dashed arrows describe the processes which occur between phases. Development proceeds in the direction of the arrows, although the activity between phases will be iterative. The solid arrows depict the CASs (defined in Chapter 1). A CAS may span several phases of the life cycle, as is evident from the figure. A brief description of the life cycle follows.

The life cycle begins when a problem is recognized by a decision maker, who then initiates a study by communicating the problem to an analyst [Balci 1987]. As opposed to the scenario given at the beginning of this chapter, formal study begins to determine what the problem actually is. *Problem Formulation* is the process of defining the problem sufficiently enough to enable specific research action [Woolley and Pidd 1981; Balci 1987]. Among the activities of the modeler during this process are attempting to justify solving the problem, determining the root causes of the problem, determining the nature of the results (prescriptive or descriptive), and determining who the decision makers are [Balci and Nance 1985]. *Formulated Problem Verification* CAS substantiates that the formulated problem entirely contains the *actual* problem [Balci and Nance 1985].

When the formulated problem is complete, *Investigation of Solution Techniques* ensues [Balci 1987]. During this process, *all* possible solution techniques are identified. As was mentioned previously, problems do not originate as "simulation problems." Simulation is merely an alternative solution for *some* problems. It might be the case that simulation is too costly or too difficult to use. Opting for simulation in such a case is absurd. As Balci [1987] points out, the goal is not just to find *a* solution to the problem, but to find an *acceptable and sufficiently credible* one. Assuming that the solution technique chosen is simulation, the *Feasibility Assessment of Simulation* CAS will justify simulation using indicators such as cost/benefits ratios, resource procurement capability, and the ability to meet requirements specifications [Balci 1987].

Balci [1987] defines a system as "any collection of interacting elements that operate to achieve some goal." To this point in the life cycle, the problem has been well-formulated but the *system* under study has not yet been defined. *System Investigation* identifies the parameters of the system (input and output variables) [Balci 1987] and the six characteristics of the system: (1) change, (2) environment, (3) counterintuitive behavior, (4) drift to low performance, (5) interdependency, and (6) organization [Shannon 1975]. The reader is referred to [Balci 1987] for a description of these characteristics with respect to the System Investigation process.

The result of system investigation is the *System and Objectives Definition* [Nance 1981], which will drive the SMDLC. The *System and Objectives Definition Verification* CAS justifies that "the system characteristics are identified and the study objectives are defined with sufficient accuracy." [Balci 1987] As Balci notes, failure to adequately perform this CAS may result in high cost of correction later in the life cycle or a type II or type III error. Upon realization of the system and objectives definition, the SMDLC is entered. The SMDLC extends from the System and Objectives Definition phase full cycle through the Model Results phase, where redefinition allows iteration through the life cycle.

Using the system and objectives definition, *Model Formulation* is performed, allowing the modeler to express his view of the model. The result of model formulation is the *Conceptual Model* [Nance 1981]. The conceptual model is a description of the model as seen by the modeler. Nance refers to this phase as the conceptual model phase because it represents the modeler's conceptualization of the model. The conceptual model should identify the model components and the component relationships, and the boundaries on model behavior. This phase parallels a similar phase of software development life cycles, where a software requirements analysis is conducted and a software requirements specification created. In addition to the above, the conceptual model must reflect the model's input data requirements and experimentation concerns. The validity of the conceptual model is assessed through the *Model Qualification* CAS. This CAS ensures that the conceptual view completely captures the system under study and has not overlooked any necessary considerations.

During *Model Representation* [Balci 1987], the conceptual model is transformed into the *Communicative Model* [Nance 1981] which "can be judged or compared against the system and the study objectives" by other humans [Nance 1981]. There may be several communicative models; some are completely different, some are refinements of previous communicative models. Balci [1987] presents a summary of representation forms that the communicative model might take. Among these are data flow diagrams, flowcharts, and

structured English and pseudocode. A similar phase in software life cycles involves refinement of the requirements specification to a preliminary design, and subsequently to a detailed design. *Communicative Model Verification* CAS "confirms the adequacy of the communicative model to provide an acceptable level of agreement for the domain of intended application." [Balci 1987] As can be seen from Figure 1, this CAS extends back through the life cycle to the system and objectives definition phase.

The *Programming* process results in the realization of a computer-executable model [Balci 1987]. The communicative model(s) (i.e., the model specification) is transformed using a (simulation) programming language into the executable *Programmed Model* [Nance 1981]. Programmed model development is guided by a software development life cycle which, like the simulation model life cycle, indicates what to do and when to do it. Overstreet et al. [1986] identify and describe seven major life cycle models proposed in the literature. It is worth noting here that, depending on the software life cycle used to guide programmed model development, the phases of the software life cycle may not "fit" exactly within the Programming process depicted in the SMDLC. For example, Boehm's Waterfall Model [Boehm 1976] calls for System Requirements, Software Requirements, and Preliminary Design phases before deriving a detailed design and beginning coding. These phases parallel the programmed model aspects of the Conceptual Model and Communicative Model phases, yet they are part of the software life cycle. This difference in "fit" is purely cosmetic and should be viewed accordingly. The *Programmed Model Verification* CAS [Balci 1987], using verification techniques well-known in the software engineering community, verifies that the programmed model is an accurate translation of the communicative model.

Combining the programmed model with the executable description of the test environment forms the *Experimental Model* [Nance 1981]. The combining of these two in a cost-effective way is the function of the *Design of Experiments* process [Balci 1987]. Balci identifies a number of techniques which help minimize cost and maximize effectiveness. *Experiment Design*

Verification CAS [Balci 1987] verifies that the experimentation plan has been accurately designed with regard to the programmed model. This verification is performed by measuring such indicators as: are the algorithms used for random variate generation theoretically accurate?, or How well are the underlying assumptions of the statistical techniques implemented to design and analyze the simulation experiments satisfied? [Balci 1987] In addition to experiment design verification, the *Model Validation* CAS [Balci 1987] is performed by comparing the experimental model with the real system (as reflected in the system and objectives definition). This CAS validates that the experimental model suitably represents system behavior and that it experiments with the model in a realistic way.

Each *Experimentation* [Balci 1987] with the model produces a single *Model Result* [Nance 1981]. Experimentation will be repeated (under different conditions) to produce as many simulation results as are necessary to derive a solution to the problem. The *Credibility Assessment of Model Results* CAS assures the quality of the experimental model with respect to the system and objectives definition [Balci 1987]. The *Redefinition* process [Balci 1987] provides a path for recycling through the entire SMDLC to allow for modification and maintenance of the model.

The CAS shown in the center of Figure 1 is *Data Validation*. Data validation is an on-going confirmation that the data used throughout model development phases are accurate, complete, unbiased, and appropriate [Balci 1987]. The two types of data that Balci classifies are model input data and model parameter data.

During the *Presentation of Model Results* process, the model results are interpreted and presented to the study sponsor for acceptance and implementation [Balci 1987]. According to Balci, this process should present the results with respect to the intended use of the model, and if necessary, the results should be integrated to support the decision maker in the decision-making process. This *Integrated Decision Support* allows recognition of model be-

havioral trends and prediction of untested behavior based on prior results [Nance 1981]. *Presentation Verification* CAS verifies the presentation of model results [Balci 1987]. Balci identifies four indicators used for verification. These indicators are: (1) interpretation of model results, (2) documentation of simulation study, (3) communication of model results, and (4) presentation technique. It is then up to the decision maker to determine the *Acceptability* (with implied implementation) *of Model Results* [Balci 1987].

The reader is referred to [Balci 1987; Nance 1981; Balci and Nance 1985] for complete discussion of the life cycle of a simulation study.

3.0 A TAXONOMY FOR PROGRAMMED MODEL VERIFICATION METHODS

The software engineering literature is laced with methodologies, tools, and techniques for designing and developing software and performing verification and validation of the resulting products. Yet, as is pointed out in previous chapters, little proliferation of these techniques to the simulation community is evident. Additionally, the simulation model has somewhat different verification needs, one of which is to show *credibility* of results. Some of the less cost effective techniques for non-simulation software become valuable PMV activities. The literature falls short when applying verification to the simulation programmed model. Again, the textbook survey of Balci [1987] and the literature review of Balci and Sargent [1984] are cited to support this claim. Table 1 (reprinted from [Balci 1987]) shows the results of Balci's survey (the PMV column corresponds to coverage given to programmed model verification). In response to this shortcoming, the research described herein was started.

PMV is concerned with the accuracy of transformation of the detailed model specification (communicative model) into the programmed model. Techniques to perform verification can be categorized by the basis with which the accuracy is justified. The taxonomy presented in this thesis categorizes the verification process into six distinct verification perspectives. These are: informal, static, dynamic, symbolic, constraint, and formal analysis. The taxonomy is shown in Figure 2. It should be noted that some of these categories are very close in nature and in fact have techniques which overlap from one category to another. There is, however,

Table 1. Topical distributions of books on discrete-event simulation methodology (reprinted from [Balci 1987])

	IM	SI	IDA	SPL	RNG	RVG	TFM	PMV	MV	SAS	MSP	CSE	other†	TNP
Neelamkavil (1987)	78 26.4%			86 28%	14 4.6%	13 4.2%	20 6.5%		15 4.9%	14 4.6%			67 21.8%	307
Ripley (1987)	41 17.3%		21 8.9%		37 15.6%	40 16.9%				50 21.1%			48 20.2%	237
Banks and Carson (1984)	46 8.9%		82 16%	35 6.8%	32 6.2%	34 6.6%	18 3.5%	6 1.2%	20 3.9%	82 16%			159 30.9%	514
Gottfried (1984)	64 21.3%			26 8.7%	25 8.3%	32 10.7%						71 23.7%	82 27.3%	300
Pidd (1984)	10 4.2%			29 12.2%	10 4.2%	11 4.6%	86 36.3%			22 9.3%			69 29.2%	237
Bratley, Fox, and Schrage (1983)	41 10.7%		20 5.2%	53 13.8%	34 8.9%	46 12%				93 24.3%			96 25.1%	383
Solomon (1983)	10 2.2%		56 12.4%	66 14.6%	33 7.3%					68 15%	3 0.7%	111 24.6%	105 23.2%	452
Bulgren (1982)	3 1.3%			83 36%	9 3.9%	20 8.7%	5 2.2%	2 0.9%	2 0.9%	17 7.4%		19 8.3%	70 30.4%	230
Law and Kelton (1982)	55 13.8%		62 15.5%	71 17.7%	19 4.7%	37 9.2%	8 2%	3 0.8%	12 3%	102 25.5%			31 7.8%	400
Mitrani (1982)	15 8.1%			48 26%	11 6%	8 4.3%				40 21.6%		23 12.4%	40 21.6%	185
Payne (1982)	26 8%		55 17%	24 7.4%	8 2.5%		39 12.1%	5 1.6%	12 3.7%	49 15%			106 32.7%	324
Rubinstein (1981)	25 9%				15 5.4%	74 26.6%				124 44.6%			40 14.4%	278
Graybeal and Pooch (1980)	14 5.6%		57 22.9%	26 10.5%	12 4.8%	8 3.2%	17 6.8%		2 0.8%	21 8.4%			92 37%	249
Maryanski (1980)	40 12.2%		9 2.7%	217 66.2%	10 3.1%	15 4.6%			5 1.5%	23 7%			9 2.7%	328
Deo (1979)	16 8%			24 12%	8 4%	7 3.5%	3 1.5%		3 1.5%	18 9%		82 41%	39 19.5%	200
Lewis and Smith (1979)	55 14%		4 1%	51 13%	28 7.2%	9 2.3%	37 9.4%	6 1.5%	4 1%	6 1.5%		24 6.1%	169 43%	393
Fishman (1978)	19 3.7%			75 14.6%	45 8.8%	88 17.1%	40 7.8%			180 35%			67 13%	514
Gordon (1978)	52 16.1%		31 9.6%	73 22.5%	6 1.9%	10 3.1%	44 13.6%			20 6.2%			88 27%	324
Lehman (1977)	20 4.9%			127 30.9%	9 2.2%	26 6.3%		5 1.2%	20 4.9%		39 9.5%	145 35.2%	20 4.9%	411
Poole and Szymankiewicz (1977)	17 5.1%			27 8.1%			16 4.8%			7 2.1%	16 4.8%	199 59.8%	51 15.3%	333
Lewis (1975)					46 30.7%	53 35.3%							51 34%	150
Shannon (1975)	46 11.9%	22 5.7%	21 5.4%	38 9.8%	10 2.6%	10 2.6%	6 1.5%		30 7.8%	58 15%	31 8%	71 18.3%	44 11.4%	387
Fishman (1973)	25 6.5%		19 4.9%	63 16.4%	27 7%	42 10.9%	43 11.2%			111 28.8%			56 14.5%	386
Maisel and Gnugnoli (1972)	35 7.5%		50 12.9%	152 32.7%	12 2.5%	10 2.1%				19 4.1%		59 12.7%	118 25.4%	465
Mihram (1972)	98 18.6%		105 20%		22 4.2%			5 0.9%	6 1.1%	256 48.7%			34 6.5%	526
Emshoff and Sisson (1970)	56 18.5%			43 14.2%	7 2.3%	6 2%	52 17.2%		3 1%	30 10%		69 22.9%	36 11.9%	302
Schmidt and Taylor (1970)	49 7.6%		20 3.1%	29 4.5%	40 6.2%	58 9%			9 1.4%	43 6.7%		150 23.3%	246 38.2%	644

† May include references, exercises, appendices, index, and other topics not directly related to discrete-event simulation methodology.

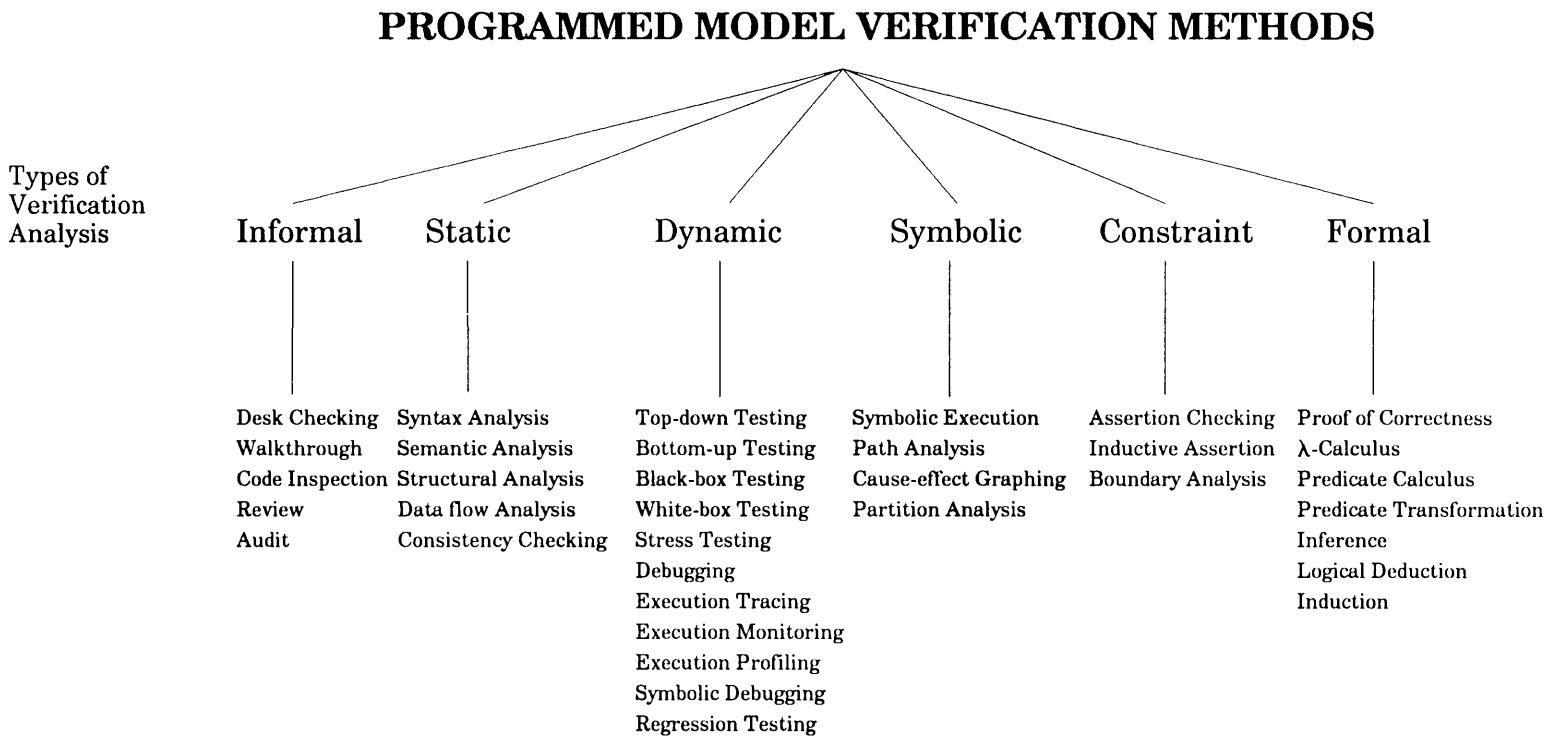


Figure 2. A Taxonomy for Programmed Model Verification Methods

a fundamental difference between each classification, as will be evident in the discussion of each.

Underneath each category, the techniques used to perform the verification are listed. The level of mathematical formality of each category continually increases from very informal on the far left to very formal on the far right. Likewise, the effectiveness of each increases from left to right. As would be expected, the complexity also increases as the method becomes more formal. The two categories, dynamic analysis and constraint analysis, are instrumentation-based, i.e., they utilize extraneous information present in the code to assist and/or enhance the analysis, particularly in an automated sense. Automated analysis usually results in higher computer resource cost but lower human resource cost.

The taxonomy provides a number of perspectives of PMV. In informal analysis, the perspective of human reasoning and subjectivity is captured. Static analysis verifies on the basis of characteristics evidenced in the code of the programmed model itself. Dynamic analysis captures the execution behavior of the programmed model, while symbolic analysis justifies the selection of the dynamic test sets and verifies the transformation of model inputs to outputs. Constraint analysis verifies conformance of the programmed model to model assumptions. Constraint analysis also serves as a validation reference by assuring that the model is functioning within the model domain. Formal analysis provides the ultimate baseline for PMV efforts.

Figure 3 summarizes a number of characteristics of the general nature of each category. These characteristics are: (1) the basis for verification, (2) the relative level of mathematical formality, (3) the complexity of the associated techniques, (4) cost in terms of human time and effort, (5) cost with respect to computer resources (e.g., execution time, memory utilization, storage requirements, etc.), (6) the relative effectiveness of the method in general, (7) the relative importance of the associated techniques to PMV, and (8) whether or not the category is considered instrumentation-based. The comparison among the categories (e.g., Level of

Category / Characteristic	Informal Analysis	Static Analysis	Dynamic Analysis	Symbolic Analysis	Constraint Analysis	Formal Analysis
Category Definition	Analyzing through the employment of in-formal design and development activities	Analyzing characteristics of the static source code	Analyzing results gathered during model execution	Analyzing the transformation of symbolic inputs to outputs along model execution paths	Comparison of actual model execution state with assumptions	Formal mathematical proof of correctness
Level of Formality	Very Informal	Informal to Formal	Informal to Formal	Formal	Formal to Very Formal	Very Formal
Complexity	Low	Moderate	Moderate to High	High	High to Very High	Very High
Human Resource	Very High	Low to Moderate	Moderate to High	High	High	Very High
Computer Resource Cost	Very Low	Moderate to High	Very High	Moderate	High	Very Low
Effectiveness	Limited	Moderate to High	Moderate to High	High to Very High	Very High	Highest, if Attainable
Instrumentation Based	No	No	Yes	No	Yes	No
Importance to PMV	High	High	High	Very High	Very High	Highest, if Attainable

Figure 3. Characteristics of the PMV Methods Under Each Category

Formality) is intended more to give a relative view among the spectrum of categories rather than to measure against some known standard. Figure 4 provides the advantages and disadvantages of each category.

In Figure 5, PMV methods are cross-referenced with the various CASs of the simulation model development life cycle under each category. All verification techniques of a particular category applicable for a CAS are listed. This figure provides quick reference of which techniques should be considered viable at the various stages of the credibility assessment.

In the typical discussion of software verification techniques, the emphasis is on the one-dimensional picture of showing program correctness. The concern is that the program works, and works correctly. Other than showing program correctness, and perhaps improving the quality of the product, the typical software engineering project may realize only moderate gains by classifying the means of verification in specialized categories, such as is in the taxonomy. The simulation study has everything to gain. By applying verification techniques categorically, the modeler not only realizes a verified model, he has categorical evidence from a broad range of verification perspectives to substantiate his claims. The taxonomy is beneficial to the modeler: (1) by categorically identifying techniques which will allow him to verify the programmed model, and (2) by guiding the modeler with an effective, well-organized format for assessing the credibility of simulation results.

3.1 A Simulation PMV Case Study

In this section, the use of the taxonomy for PMV is illustrated with a simulation case study. A description of the simulation problem is given, from which model specifications to be converted into a programmed model can be derived. From the taxonomy, a number of verification

Category	Informal Analysis	Static Analysis	Dynamic Analysis	Symbolic Analysis	Constraint Analysis	Formal Analysis
ADVANTAGES	Allows human reasoning and consideration of subjective aspects. Techniques cover a broad range of the life cycle. Catches errors in the early phases of development. Provides useful documentation of the development process. Has a very low computer resource cost. [Yourdon 1985; Fagan 1976; Bryan and Siegel 1987]	Well-defined process with many tools available. Highly automated, resulting in low human resource cost. Enhances the ability to perform other types of analysis. Provides extensive documentation of model structure, logic, and data flow. [Fairley 1976; Ramamoorthy and Ho 1977; Westley 1979; Allen and Cocke 1976; Sneed and Merey 1985]	Allows observation of model execution behavior--is the only way to "see" how the model functions, particularly with respect to given hardware. Highly automated, with many tools available. Provides extensive documentation of model behavior. [Deutsch 1982; Myers 1979; Fairley 1976; Lauesen 1979]	Provides comprehensive path coverage which subsequently aids test data generation. Aids the verification of classes of input data, further aiding the generation of test data. Provides an excellent source of documentation by completely depicting possible execution outcome. [Osterweil 1983; Clarke and Richardson 1983; Richardson and Clarke 1985; Clarke 1976; Myers 1979; Chusho 1987; Ramamoorthy, et al. 1976]	Very powerful techniques for assuring model assumptions are not violated. Provides a means for inductively proving model correctness. Aids in the recognition of "interesting" test situations. Provides useful "inline" documentation of model assumptions. [Myers 1979; King 1976; Miller 1977; Stucki 1977; Stucki and Foshee 1975; Saib, et al. 1977; Buckles and Ryan 1977]	Provides the highest level of verification possible. [Backhouse 1986; King 1976; Clarke 1988; Hoare 1969; Miller 1977]
DISADVANTAGES	Requires manual activity with associated high human resource cost. Prone to human error. Success depends on the expertise of the individual. [Balci 1987; Yourdon 1985; Adrion, et al. 1982; Westley 1979]	Limited scope of verification: cannot show correctness, cannot verify modeler intentions, cannot determine execution behavior. Tools used to perform the verification must themselves be verified (e.g., the compiler must be correct). [Adrion, et al. 1982; Fairley 1978; Westley 1979]	Cannot show correctness: can only verify individual model runs. Difficult to obtain adequate execution coverage, i.e., determining what to test and how to test it. Obtaining execution data degrades performance. Test planning, test data generation, and analysis of results often difficult and time consuming. [Cherniavsky and Smith 1987; Deutsch 1982; Berg, et al. 1982; Prather and Myers 1987; Davis and Weyuker 1988; Miller 1977]	Symbolic expressions are difficult to derive. Complexity grows in exponential fashion as the model grows. Has limited use with complex data structures. Cannot represent looping structures. [Ramamoorthy, et al. 1976; Clarke 1976; Myers 1979; Westley 1979; Chusho 1987; Howden 1977]	Creating the necessary formal specifications is difficult. High human resource cost involved. Performing techniques during execution degrades model performance. [Myers 1979; Saib, et al. 1977; Buckles and Ryan 1977; Stucki and Foshee 1975]	Far too complex to be considered a realistic method of model verification. Difficulties arise when dealing with hardware idiosynchrosies such as round-off or floating point representation. [Ramamoorthy, et al. 1976; King 1976; Miller 1977; Hoare 1969]

Figure 4. Advantages and Disadvantages of PMV Methods

Category / CAS	Informal Analysis	Static Analysis	Dynamic Analysis	Symbolic Analysis	Constraint Analysis	Formal Analysis
Model Qualification	Desk Checking Walkthrough Review Audit	Structural Analysis Data Flow Analysis Consistency Checking				
Communicative Model Verification	All methods applicable	Semantic Analysis Structural Analysis Data Flow Analysis Consistency Checking	White-box Testing Regression Testing (Top-down, Bottom-up, and Black-box Testing are also included as part of test planning)	All methods applicable	All methods applicable	All methods applicable
Programmed Model Verification	All methods applicable	All methods applicable	All methods applicable	All methods applicable	All methods applicable	All methods applicable
Experiment Design Verification	Desk Checking Walkthrough Review Audit	Semantic Analysis Structural Analysis Data Flow Analysis Consistency Checking	Black-box Testing White-box Testing Stress Testing Execution Tracing Execution Monitoring Execution Profiling Regression Testing	Cause-effect Graphing	Assertion Checking Boundary Analysis	All methods applicable

Figure 5. PMV Methods Applicable to Some Other CAS

techniques from various categories are selected and verification is performed on selected model components. Justification is given for the credibility of the simulation results.

The purpose of the illustration is twofold. First, the illustration directly points out the usefulness of the taxonomy for guiding the modeler through a broad-based verification process. Secondly, the actual use of verification techniques is illustrated. A detailed description of each technique listed in the taxonomy is given in Chapter 4. The reader is advised to consult that chapter for information concerning each particular technique used.

3.1.1 Case Study Problem Description

The following problem description for this case study is taken from [Balci 1988].

A batch computer system operates on two processors. Jobs arriving to the system are processed by a job entry scheduler (JES). Following processing by the JES, the job will be scheduled on either CPU 1 or CPU 2. Following CPU processing, the job will either send output to the system printer (PRT) and depart the system, or simply depart the system. A general picture of system operations is given in Figure 6 as reprinted from [Balci 1988].

The users of the system are classified into 4 categories: (1) users dialed-in by using a modem with 300 baud rate, (2) users dialed-in using a 1200-baud modem, (3) users dialed-in using a 2400-baud modem, and (4) users connected to a 9600-baud local area network (LAN). Each user develops his own batch program and submits it to the system for processing. Based on collected data, interarrival times of batch programs to the system with respect to each user type are determined to have an exponential probability distribution as shown in the top part of Figure 7.

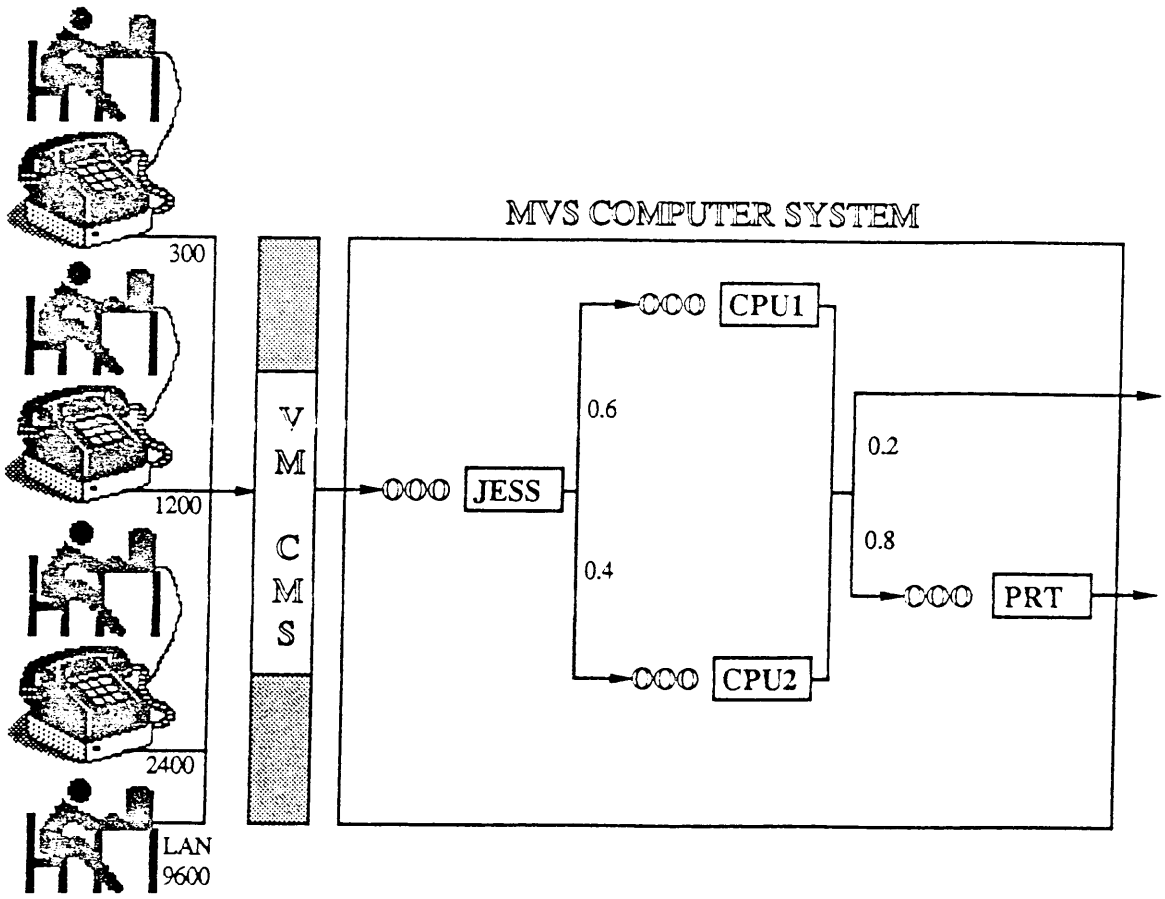


Figure 6. Description of a Simulation Case Study (reprinted from [Balci 1988])

Type of User	Interarrival Times	Mean
Modem-300 User	Exponential	3200 seconds
Modem-1200 User	Exponential	640 seconds
Modem-2400 User	Exponential	1600 seconds
LAN-960 User	Exponential	266.67 seconds

Facility	Processing Times	Mean
JES Scheduler	Exponential	112 seconds
Processor 1 (CPU 1)	Exponential	226.67 seconds
Processor 2 (CPU 2)	Exponential	300 seconds
Printer	Exponential	160 seconds

Figure 7. Probabilistic Characterization of Interarrival Times and Processing Times

When a batch program is submitted, it first goes to the JES. The job is considered to have entered the system at the time at which it arrives in the queue for the JES. The job is assigned to CPU 1 by the JES with a probability of 0.6 or to CPU 2 with a probability of 0.4. At the completion of program execution, the program's output is sent to the system printer with a probability of 0.8. The probability is 0.2 that the job departs the system immediately following program execution.

All queues are managed on a first come, first served basis, with each facility (JES, CPU 1, CPU 2, or PRT) processing only one job at a time. The probability distribution of the processing times spent for each program on a given facility is given in the bottom part of Figure 7.

The model is to simulate the behavior of the system, and construct confidence intervals for the following measures of interest:

- Utilization of the Job Entry Scheduler (JES)
- Utilization of CPU 1
- Utilization of CPU 2
- Utilization of the printer (PRT)
- Expected time spent by a job in the system.
- Expected number of jobs in the system.

Through analysis, it is determined that the simulation model reaches its steady state conditions after 3,000 jobs have departed the system. The system is then to be simulated in steady state for 15,000 jobs departing the system. In order to construct the confidence intervals, it is determined to replicate the simulation run 20 times.

The model is to be implemented using the event scheduling world view [Balci 1988]. A flowchart to implement this world view, adapted to this case study, is given in Figure 8. The event scheduling world view sees model activity as a series of events which alter the state of the

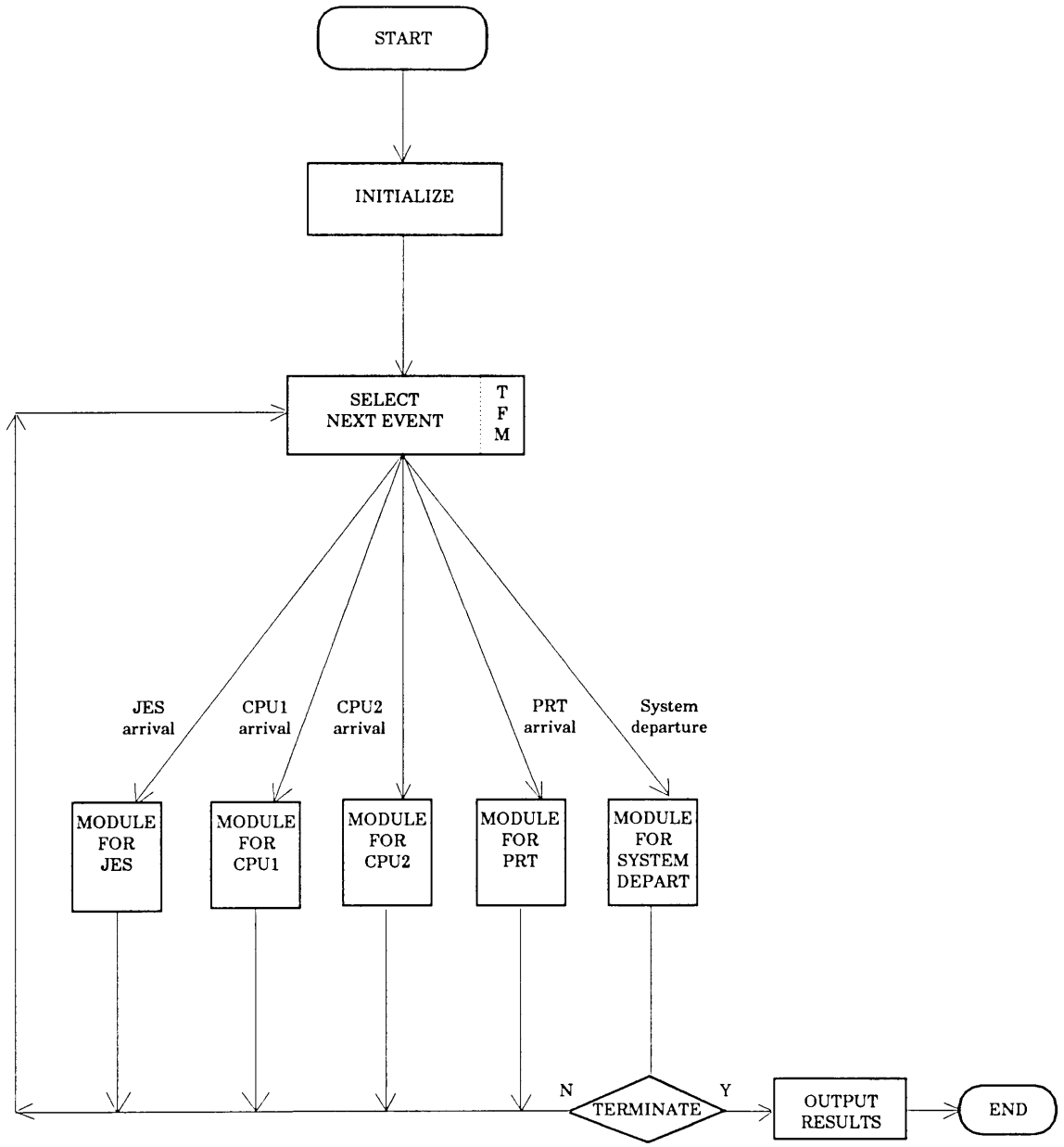


Figure 8. Flowchart for the Event Scheduling World View

model. The model emphasizes the scheduling of these events at particular points in time. The events defined in this model are:

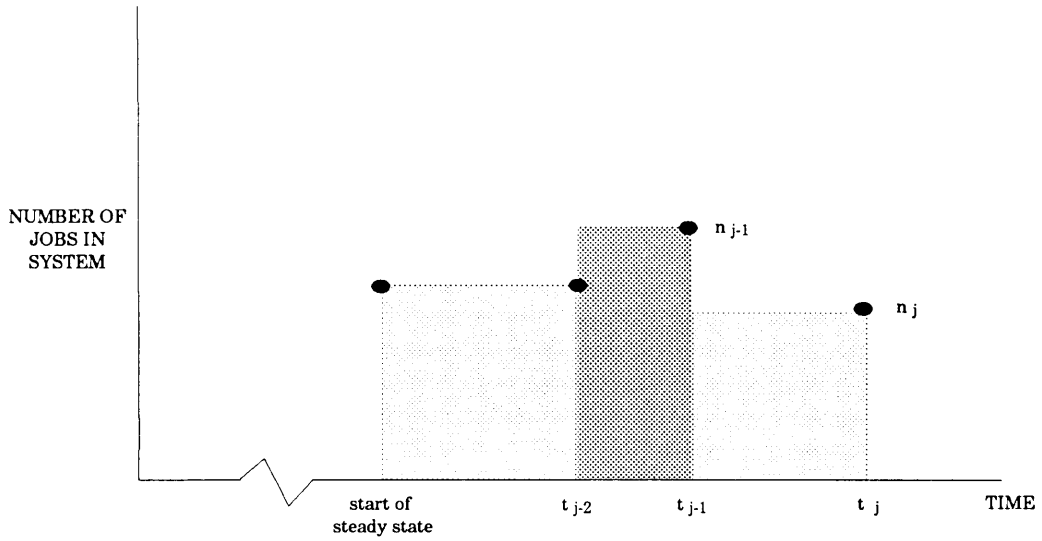
1. An arrival to the JES.
2. An arrival to CPU 1.
3. An arrival to CPU 2.
4. An arrival to PRT.
5. A departure from the system (i.e., an arrival to the outside of the system).

One replication of the simulation involves initializing the model, selecting the next event from the head of a list of events, determining what the next event is, performing the processing for that event, and then selecting the next event from the list, etc. Termination of this simulation is strictly based on the number of jobs departing the system.

Time is simulated via a system clock which is updated to match the time of the event at the head of the list. The list is maintained in ascending order of the scheduled time of the events in the list. The list is often referred to as a future events list because the majority of the events in the list are scheduled to occur at some future time during the simulation. The clock to be used in this model is a variable time flow mechanism (TFM) because it is updated in *variable* time increments.

Among the performance measures of interest is the expected number of jobs in the system. This result depends on the accumulated time of all jobs in the system during the course of the steady state simulation. The accumulated time can be graphically represented as an area, as shown in the graph in Figure 9. This area is accumulated each time the number of jobs in the system changes, i.e., at each new JES arrival or system departure event, as follows:

1. Determine the amount of time since that last arrival or departure event.
2. Multiply the time by the number of jobs in the system before the change occurred.



j	arrival or departure event
j-1	last arrival or departure event
n_j	number of jobs in system at time t_j
n_{j-1}	number of jobs in system at time t_{j-1}
t_j	current time (i.e., clock)
t_{j-1}	time of last arrival or departure event

Figure 9. Graph Depicting the Computation of AREA

3. Add the result to the previous total to give the new total.

The programmed model computation of this sum will be verified below.

When verifying the model, one must reference the model specification. The PMV is, after all, the reflection of how accurately the programmed model has been transformed from its specification. The event scheduling world view flowchart in Figure 8 is one specification of the model. The model description is another specification. By performing structural analysis—a static analysis technique—on the programmed model, the similarity between the world view flowchart specification and the programmed model can be seen (see Figure 10). From a structural viewpoint, we can see an accurate translation of the specification.

Another specification is given below:

The result of processing a JES arrival is:

1. The total jobs in the system is increased by 1.
2. Another job, originating from the same input device as the current job being serviced, is scheduled to arrive at the JES at some point in the future.
3. The current job is serviced by the JES when the JES is available, and is scheduled for servicing by either CPU 1 or CPU 2.
4. The JES ensures that only the current job is being serviced by the JES at the time of its servicing.
5. Steady-state JES utilization time is recorded.

The JES determines its the duration of service time using an exponential distribution with mean of 112 seconds.

During JES processing, the CPU to be scheduled is determined with 60% probability of CPU 1 being selected and 40% probability of CPU 2 being selected.

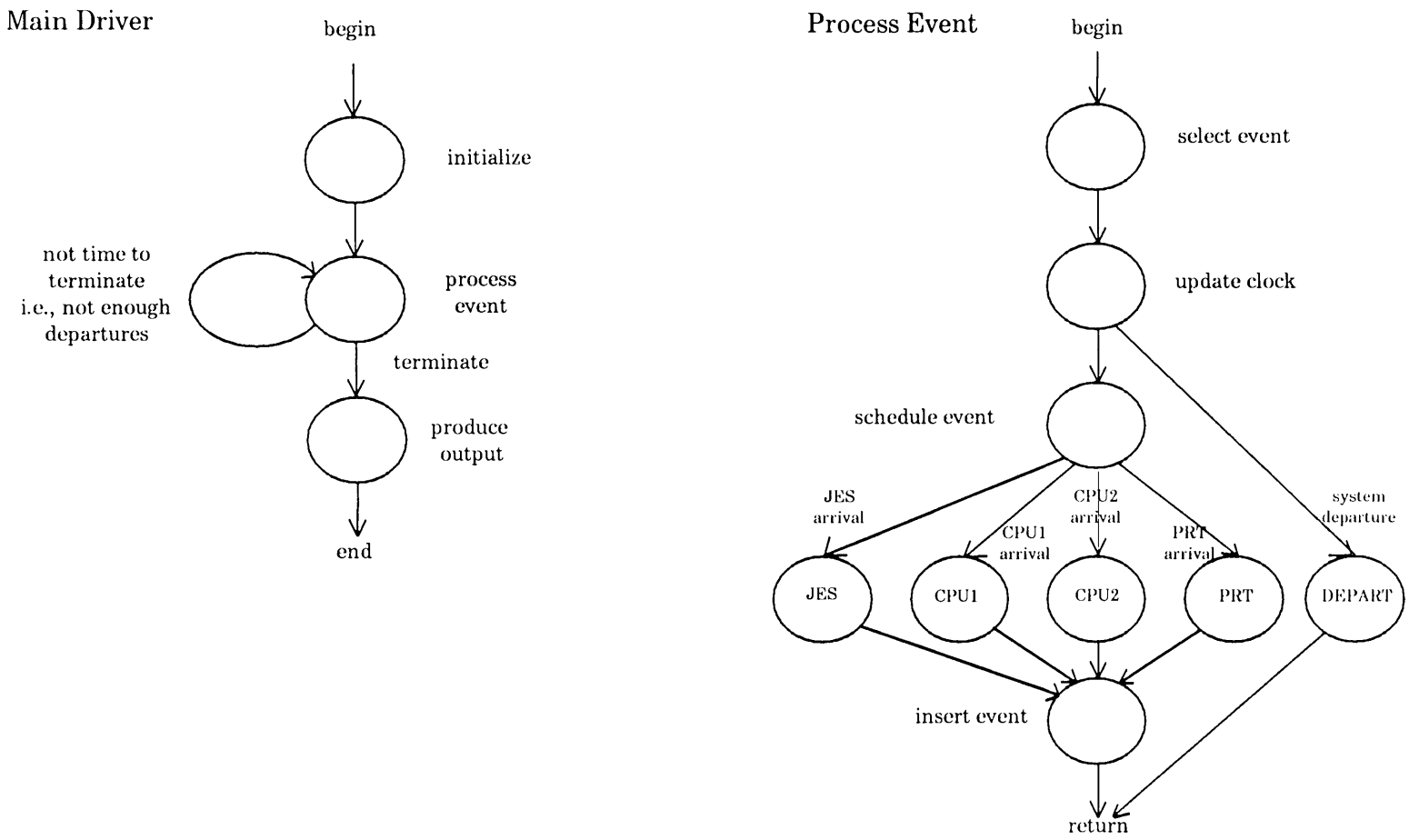


Figure 10. Programmed Model Flowchart

A means of verifying this portion of the programmed model would be to show that the outcome of executing the segment of the model that processes JES arrival events matches the stated requirements. This technique is known as black-box, or functional, testing.

The programmed model for this case study is shown in Appendix A. This listing was created via a source code formatter (a.k.a. "pretty printer") which enhances the listing with lines demarcating the model block structures. Static analysis techniques were used to create the Cross-Reference Report in Appendix B, the Identifier Report in Appendix C, the Hierarchy Report in Appendix D, the Warning Report in Appendix E, the Duplicate Identifier Report in Appendix F, the Side Effects Report in Appendix G, the Totals Report in Appendix H, and the Sorted Procedure Index Report in Appendix I. The programmed model was instrumented to allow model execution to be profiled. A model profile can be seen in Appendix J. This particular profile indicates the number of times each line was executed during a simulation. Figure 11 illustrates a debugging session using a symbolic debugger. In addition to providing extensive documentation about the model, the listings were indispensable for performing the verification described below.

3.1.2 A Case Study Verification Example

The specification for computing the accumulated time—or ***area***—used to compute the expected number of jobs in the system was given above. In order to verify the simulation result for expected jobs in the system, it is necessary to verify that ***area*** is being accumulated properly. This is done by utilizing techniques from the static, dynamic, symbolic, and constraint analysis categories of the taxonomy.

The JES arrival and system departure events are the only two events which are to alter the number of jobs in the system. A JES arrival corresponds with the entry of a job to the JES queue. When a JES event occurs, the accumulated time since the last change in number of

```
591       (* Schedule the first job from each entry device *)
592       (*************************************************)
593       for device := M300 to LAN do
594          ARRIVAL(device,event_list,system_data);
595
596       (*************************************************)
597       (* Process the specified number of transient and *)
598       (* steady-state jobs.                            *)
599       (*************************************************)
600       while system_data.completed_jobs < steady_state_len do
601          PROCESS_EVENT(event_list,system_data);
602
603       WRITE_RESULTS(system_data)
604    end
605 END.  (* program SIMULATION *)
──D:\RICK\THESIS\EVENTSCH.PAS──────────────────────────────────TDEBUG 4.01──
1 $4990:$0012 event_list^.next_event        PRNT_ARRIVAL 3
2 $4556:$000B system_data.clock             2396.9638        2.3969637621E+03
3 $4556:$0005 system_data.completed_jobs    $0006  00000000 00000110  6
4 $4556:$0017 system_data.time_j_minus_1    0.0000         0.0000000000E+00
5 $4556:$0009 system_data.seed              $6F03  01101111 00000011  28419
──Watch──────────────────────────────────────────────────────────────────────
* N
Break at : D:\RICK\THESIS\EVENTSCH.PAS\600  $430B:$0C8D
*
```

Figure 11. Debugging Session with a Symbolic Debugger

jobs (i.e., the last arrival or departure event) must be summed. Likewise, a system departure signals a job leaving the system and the need to update the sum. Thus we would first like to verify that the sum is altered only in these two situations.

The function AREA_J (line 259 in Appendix A) in the programmed model computes the sum of variable ***area***. By placing assertion statements in AREA_J as shown in Figure 12, we can ascertain that AREA_J was only called when the current event was either a JES arrival or a system departure. Further, we wish to verify that AREA_J was only called when the system was in the steady state. This can also be accomplished with an assertion statement.

Another alternative exists. The model profile in Appendix J and cross-reference report in Appendix B answer both questions concerning the state of the model when AREA_J was called. Using the cross-reference report, we know AREA_J is only referenced (called) from lines 345 and 453 of the model. Looking at those two lines in the profile, we see 14,995 and 15,000, or a total of 29,995, calls were made to AREA_J, respectively. Line 259 verifies that 29,995 was the sum of all calls made to AREA_J. Line 345 resides in procedure ARRIVAL which, with the exception of the initialization of jobs in line 594, is only excuted when ***next_event*** is a JES arrival. Similarly, line 453 resides in procedure DEPARTURE, which is only executed when ***next_event*** is a system departure. Lines 345 and 453 follow a predicate governing when the two statements can be executed. ***in_steady_state*** is a boolean variable which is originally set to false in line 151, in procedure INITIALIZE. ***in_steady_state*** is equivalent to the predicate ***completed_jobs > transient_state_length***. It is only altered at one place in the program, in line 459, where it is set to *true* as a result of the predicate being true. From the execution profile we can prove that line 459 is executed one time, exactly when the last transient state job left the system (i.e., after 3,000 jobs). We conclude that ***area*** is updated only when the system is in the proper state. (Although the profile reflects only the results of one particular execution of the model, other factors lead us to believe our conclusion is valid, such as the structure of the predicates governing entry into the steady state.)

```
procedure AREA_J(var system_data : system_description);

    begin

        with system_data do begin

            (* assertion check to assure the proper system state *)
            if not (in_steady_state AND
                (current_event in [JES_arrival,system_departure])) then
                ASSERTION_VIOLATION(in_AREA_J);

            area := area + jobs_time_j_minus_1 * (clock - time_j_minus_1);
            jobs_time_j_minus_1 := current_jobs;
            time_j_minus_1      := clock;

        end

    end;  (* asserted procedure AREA_J *)
```

Note that variable current_event must be declared, and must be kept updated with each new event. This can be done immediately following selection of the next job in line 471. Additionally, procedure ASSERTION_VIOLATION and enumerated data type in_AREA_J must exist.

Figure 12. Asserted Submodel AREA_J

The next step in the verification process is to show that the computation of ***area*** is correct. ***area*** depends on its previous value and the values of variables ***jobs_time_j_minus_1, clock,*** and ***time_j_minus_1.*** All of these variables have appropriate initialization values upon entering the steady state. When the steady state is entered, ***area*** still holds its initial value of 0.0 (from line 158); ***jobs_time_j_minus_1*** is immediately initialized to the value of the current number of jobs in the system, ***current_jobs***. ***current_jobs*** can be shown to be correctly updated during JES-arrival and system-departure events (although it will not be shown here). Therefore we conclude that ***jobs_time_j_minus_1*** is initialized properly. ***time_j_minus_1***, upon entering the steady state, is immediately set to the current clock value. (The submodel which manages entry into the steady state can be found in lines 458 - 464.) The value for ***clock*** is dependent on the correctness of the time flow mechanism, which will be dealt with below.

With the exception of ***clock***, none of the variables are altered during steady state at any location other than in AREA_J. This invariant—the fact that the variables are altered at only this one place in the model—provides a basis for using inductive reasoning. We will focus our attention on what takes place in AREA_J. Immediately following the computation of ***area***, ***jobs_time_j_minus_1*** is set to the value of ***current_jobs***. This establishes the correct value for the "previous number of jobs" to be used the next time a job enters or leaves the system. Similarly, ***time_j_minus_1*** is immediately set to ***clock*** to establish the time when the last job entered or left the system. Inductively, we reason that the computations of ***jobs_time_j_minus_1*** and ***time_j_minus_1*** are correct.

The remaining element, ***clock***, relies on the proper sequencing of events. Events are placed on the events list in order of ascending time. The order of events must always be increasing, otherwise the simulation reverts in time. Thus the relation between ***clock*** and the scheduled time of the event at the head of the events list must be ***clock*** $\leq$ ***next_event_time***. We can simplify the verification of this relationship by noting the fact that ***clock*** is *directly* updated from the scheduled time at the head of the events list (line 475), i.e., ***clock*** is set to the scheduled

time of the next event to process, before processing the event. We need now only show that the events list is being managed properly (i.e., time is always progressing). Particularly, we want to show that the ***next_event_time*** of any job placed on the list is never less than the ***next_event_time*** of any job previously removed from the list. (Note that a job is removed from the list each time its next event is processed, and re-inserted into the list, if necessary, after processing.)

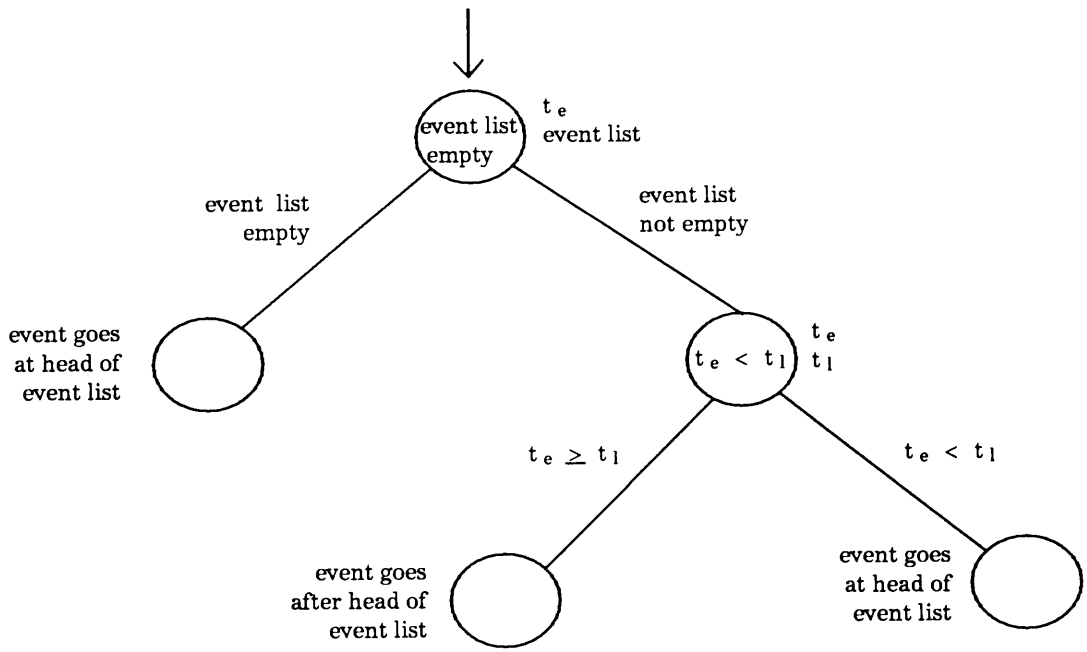
We will begin by verifying that the INSERT mechanism is working correctly. Each call to INSERT passes an event with associated ***next_event_time*** and the events list. A symbolic execution of a single execution of INSERT yields the execution tree in Figure 13. From the execution tree, which shows all possible outcomes of INSERT, we verify that the list will be ordered by increasing ***next_event_time***. Further, we can verify that the first come, first served discipline is being implemented.

We still have not verified the correctness of the time flow mechanism. We must still show that the ***next_event_time*** of any job being inserted into the list is not less than ***clock***. (Recall that ***clock*** is set to the current job's ***next_event_time*** prior to processing the event. ***clock*** will be used here to provide distinction between the current ***next_event_time*** and the ***next_event_time*** at the head of the events list.) We will show here that ***next_event_time*** $\geq$ ***clock*** is always true.

From the cross-reference report we know both locations where ***next_event_time*** is set. In ARRIVAL, we see that

```
next_event_time := EXPON + clock
```

Clearly, if ***EXPON*** ≥ 0, then ***next_event_time*** $\geq$ ***clock***. ***EXPON*** can be verified during a simulation using an assertion as in Figure 14. The other location where ***next_event_time*** is set is in procedure SCHEDULE. We verify ***next_event_time*** $\geq$ ***clock*** in SCHEDULE always holds via



t_e = next event time of event to be inserted

t_l = next event time of event at the head of the event list

Figure 13. Symbolic Execution Tree for Submodel INSERT

```
function EXPON(mean : time; var seed : integer) : time;

    var
        r, ex : time;

    begin

        EXPON := - mean * LN(RAND(seed));
        if EXPON < 0 then
            ASSERTION_VIOLATION(in_EXPON);

    end;  (* asserted function EXPON *)
```

Figure 14. Use of Assertion in EXPON

the symbolic execution tree in Figure 15, where we show that ***next_event_time*** is either dependent on ***clock***, or dependent on a value which is greater than ***clock***. (Note that ***length_of_event*** $\geq$ ***0*** is easily verified.) We conclude that clock is computed properly.

It has been shown that ***area*** is only modified during steady state at the occurrence of a JES arrival or a system departure. It has been shown that the computation of ***area*** is based on accurate parameters which are updated appropriately during execution. In addition—and as a nice side-effect—we have verified the TFM and the ordering of the events list. We also have thorough documentation from a broad range of sources. Though, we are far from formal proof of correctness, we have gathered substantial evidence with which we can assess the credibility of these portions of the model.

The taxonomy provides guidance for performing PMV by displaying a broad range of techniques logically related in six distinct categories. The categorical representation of each class's characteristics provides further insight into what will be involved when applying a technique. This is witnessed by the example above. There are other dimensions of this case study which were served through the taxonomy but not illustrated above. Informal analysis techniques allowed changes to be made to the model at the design level, before the design and related problems became "fixed in concrete." Some of the changes helped improve the quality (and credibility) of the model from a subjective viewpoint (e.g., improved representation, increased performance, etc.). Bottom-up testing of INSERT, ARRIVAL, and DEPARTURE was done by creating test harnesses to drive the execution of the procedures. Tracing, profiling, and monitoring were all performed (see the appendixes), and debugging was aided through the use of a symbolic debugger. It is evident that the taxonomy is a valuable resource for PMV. Descriptions of software verification techniques applicable for PMV are given in Chapter 4 next.

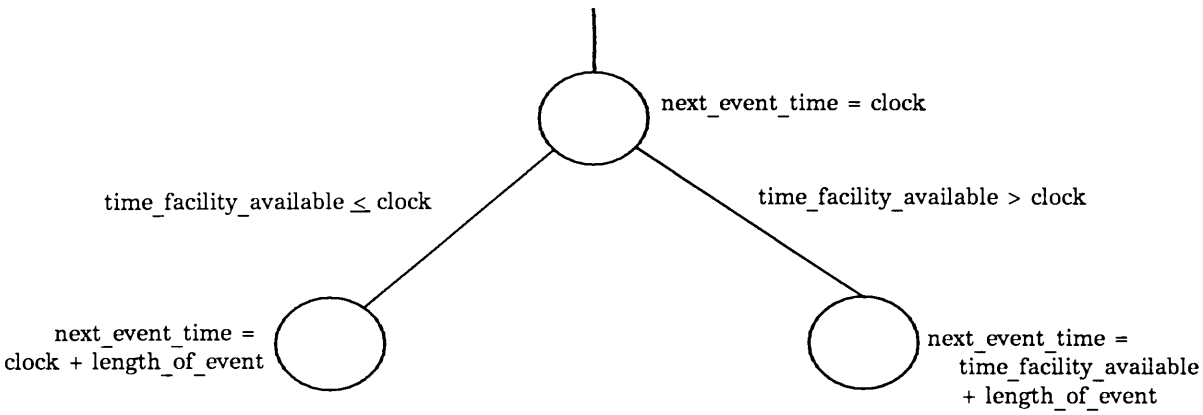


Figure 15. Symbolic Execution Tree for Submodel SCHEDULE

4.0 METHODS FOR PROGRAMMED MODEL VERIFICATION

In this chapter, each category in the taxonomy is discussed in detail. The basis for each type of verification is discussed, and techniques to perform the verification are presented. Advantages and disadvantages of each are cited. Emphasis is given to presenting the material in the context of its significance to the Simulation Model Development Life Cycle, and more specifically, to Programmed Model Verification.

4.1 Informal Analysis

Informal analysis techniques are among the most commonly used verification strategies. Verification by informal analysis is based on the employment of informal design and development activities. This category of analysis is referred to as informal because the tools and techniques used rely heavily on human reasoning and subjectivity without stringent mathematical formalism—not because of any lack of structure and formal guidelines for the use of the techniques. The informal analysis approach is a very intuitive one.

Informal analysis involves evaluation of the model using the human mind. This can be done by the modeler, a modeling team, a multidisciplinary study development group, or an independent testing organization. It includes not only evaluating the resulting model for completeness, consistency, and unambiguity of translation, but also seeks justification for the various design and development decisions made. The evaluations can be made by mentally exercising the model, reviewing the logic behind the algorithms and decisions, and examining the effects the various implementations will have on the overall outcome of the model.

Because human reasoning is involved, informal analysis can provide a broad range of coverage, simultaneously considering many dimensions of the study. For example, suppose a particular algorithm is employed to generate random variates for a part of the programmed model. The algorithm is fast and is known to be accurate. Through informal analysis, however, it may be determined that the algorithm makes horrendous use of memory, making its use unacceptable for the simulation model. Here the dimensions of execution speed, correctness and resource utilization within the range of the given hardware are all being considered together.

As another example, suppose a portion of an animated simulation is designed. In his desire to be creative, the modeler designs a very colorful and detailed display which runs quickly and with low resource utilization. Upon review, however, it may be determined that the design is ergonomically unsatisfactory. The extensive use of color and detail detract from the information that is supposed to be conveyed. Further, it may be determined that to realize the design would require coding practices that are far too complex and unmanageable. Multiple dimensions, even very subjective ones, can be captured through informal analysis. Several informal analysis techniques are discussed below.

4.1.1 Desk Checking

Desk checking is probably the most commonly used verification technique. Simply put, desk checking is the process of reviewing one's work to check its logic, consistency and completeness. Desk checking is particularly useful in the early stages of design, before the task becomes unmanageable. Most modelers perform a version of desk checking as they develop their model and then examine it to see why it doesn't work. To be truly effective, desk checking should follow tighter guidelines than this.

First of all, desk checking should be performed before the model is tested. What this means is that desk checking is not an execution debugging technique. Before energy is expended getting a model into execution, it should be thoroughly desk checked.

Secondly, desk checking should be performed by a second party [Adrion et al. 1982]. This enhances the completeness and reliability of the technique simply because the modeler often becomes blinded to his own mistakes. The second party is much more likely to detect subtle errors.

The major obstacle to performing extensive desk checking is reluctance on the part of the modeler to use it. This is because of the large investment in time that desk checking is perceived to require. The modeler is much more anxious to get his design into execution than to write and review code on paper first. Unfortunately, the long term results are usually predictable, typically with much more time being spent later uncovering simple flaws in the design that have mushroomed into larger problems. Simultaneously managing the keyboard, the text editor and the model coding process (as many modelers no doubt do) is less effective than the singular tasks of design, coding, desk checking, and keyboarding—in that order.

4.1.2 Walkthrough

Walkthroughs are a more formal approach to verification than desk checking. The walkthrough is similar to desk checking in that the design and character of the model's code are examined in detail. The logic of the model is analyzed, its consistency is verified, and its completeness determined. In an organized manner, the examiners walk through the details of the design or source code to perform the verification; hence the term walkthrough.

Unlike the loose structure of desk checking, the walkthrough is carried out under specific guidelines. It is an organized activity of the modeling organization. There are many terms associated with the concept of the walkthrough. Among such terms are code inspections, reviews, and audits, each of which are discussed as separate activities in later sections. The term *walkthrough* itself has been related to a variety of verification techniques, few of which have attained any measure of standardization. The exception is the *structured walkthrough* introduced by Yourdon [1985]. The structured walkthrough is what is discussed below.

The walkthrough is carried out by a team of individuals associated with the development process. The intent is to review and discuss the model in an effort to locate flaws in the design and/or source code. The model in review can be a high-level specification, a detailed design, or even an actual coded submodel of the programmed model. The walkthrough itself is the meeting of the team members.

The walkthrough team is composed of the modeler and study peers, most of whom are in some way familiar with and related to the simulation study. The walkthrough is a fact-finding venture. Its outcome is intended to help the subsequent development and verification of the model. It is not a forum for rating modeler performance. As such, managers should be excluded from the activities of the walkthrough. (The *review*, described later, opens avenues for

managerial involvement.) Either the manager or a member of the simulation project will establish the walkthrough team, depending on the project organization.

Yourdon identifies several roles in a structured walkthrough. They are: (1) the presenter, who most often is the modeler; (2) the coordinator, who organizes, moderates, and follows up the walkthrough activities; (3) the scribe, who documents the events of the meeting; (4) the maintenance oracle, whose responsibility is to consider long-term implications of the model; (5) the standards bearer, who is concerned with adherance to standards; (6) a user representative to reflect the needs and concerns of the sponsor; and (7) other reviewers as desired to give general opinions of the model (e.g., an auditor). Though Yourdon specifies the several roles, many authors realize a workable group of as few as three members [DeMarco 1979; Deutsch 1982; Adrion et al. 1982; Myers 1978,1979].

Before the meeting, the coordinator assures that the team members have all materials necessary. The members study the materials prior to the walkthrough. During the meeting the presenter leads the other members through the model. The model is typically "executed" by the walkthrough team using a set of prepared test cases. The content and functionality of the model are presented and the reviewers provide constructive criticism. The source code is examined for correctness, style, and efficiency. Comments are made only to the point of identifying errors and questionable practices. It is the responsibility of the modeler to digest these comments with an open mind and later seek to resolve the issues. The events of the meeting are documented and maintained as part of the on-going study documentation. As necessary, the modeler cycles back through the development process and at some point in the future, reschedules another walkthrough.

The walkthrough provides several benefits to PMV. The first is early detection of errors. This leads to higher quality and reduced development cost. It is a well-known fact among the software engineering community that the cost of error correction grows dramatically as the

development progresses. Another benefit is the documentation produced. The walkthrough documentation is useful for tracking development progress as well as for depicting model design and fundamental assumptions. A third, and far-reaching, result of the walkthrough is the dissemination of information among study members. The effects of this are several. The immediate effect is to distribute the sense of responsibility for study success from the one to the many. In the ideal sense, peer pressure obligates each to do his part to maintain excellence. The likelihood of someone recognizing and helping to remove development slack is increased. Another effect is the sharing of technical information and expertise among members. This effect has obvious merits. Still another benefit is the insurance provided. Should a team member unexpectedly leave the study in midstream, chances are good that a significant portion of his work can be salvaged. All of these elements combine for improved quality and increased likelihood of successful simulation, both in the present and in the future.

4.1.3 Code Inspection

Code inspections were introduced by Michael Fagan [1976] as an alternative to walkthroughs. The code inspection is intended to be a more formalized approach to reducing errors in model development. To a large degree, the code inspection has obtained more standardization that the walkthrough. Its sole primary purpose, as Dobbins [1987] states, is to remove defects as early in the development process as possible. Defects are to be identified and their existence and nature documented. Dobbins goes on to point out that there are several secondary purposes of the inspection process, among which are to provide traceability of requirements to design, increase model quality, reduce development cost, and improve the effectiveness of other aspects of the model life cycle. These are, according to Dobbins, "all part of the effect of performing inspections properly and professionally."

Buck and Dobbins [1983] identify three levels of the development process during which inspections are to be performed. These are at the high level design (Communicative Model Phase), the low level design just prior to coding the model, and after coding when a clean compilation has taken place (prior to testing). These levels correspond to the I_0, I_1, and I_2 inspections laid out by Fagan's earlier work. It is significant to note that, along the same vein as other informal analysis techniques, code inspections preceed testing activities.

Fagan originally specified five distinct inspection phases: overview, preparation, inspection, rework, and follow-up. The inspection process has been refined and streamlined over the years, but basically the phases are the same. Only the planning phase, prior to the overview, has been added to the process [Dobbins 1987; Ackerman et al. 1983].

The inspection team is comprised of members who play particular roles. The *moderator* manages the team and provides leadership. The moderator is responsible for all meeting logistics and coordinates activities during the meeting. The *designer* is the developer (modeler) responsible for producing the program design, while the *coder/implementor* is the programmer (modeler) responsible for translating the design to code. The *tester* is responsible for the testing activities of the model. Although four members has been found to be a workable team size, the team may have more members.

The logistics of the entire inspection process are established during the planning phase. At this point the moderator confirms the inspection team, assures adequate materials are available for members, reserves the inspection location, establishes the inspection schedule, and notifies team members.

During the overview the designer gives a brief description of the (sub)model to be inspected. The model's purpose, logic, interfaces, etc. are introduced and necessary documentation distributed to team members to study. With the notification of the inspection meeting, the

preparation phase begins. Time is given for the members to study the materials and prepare for their roles in the upcoming meeting.

The inspection meeting follows an established agenda, conducted by the moderator. Following introductions, a designated *reader* narrates the design as expressed by the designer. The purpose of the reading is to identify and discuss previously undetected defects. Errors detected are documented and classified according to their nature and severity. Care must be taken during the inspection to keep the discussion on an impersonal level and the meeting conducted in a professional manner. This is the responsibility of the moderator. Also the responsibility of the moderator is to prepare a written report detailing the events of the meeting (to be done within a day following the inspection) and to insure appropriate measures are taken in subsequent phases of the inspection process.

The designer or coder/implementor resolves problems during the rework phase and, if necessary, re-inspection takes place. The follow-up phase is completed by the moderator to assure that all defects have been corrected and the results documented. Usually there is a specifically defined exit criteria which must be met.

A key factor in the success of the inspection process is the education of the team members in the guidelines and expectations of the process. The code inspection is intended to be a more rigorous alternative to the walkthrough, accomplishing this end primarily because the process is well-defined and to a certain extent, standardized. With the increased formality, inspections tend to vary less and produce more repeatable results. Like the walkthrough, the code inspection is effective for early error correction, provides an excellent source of documentation, and removes responsibility for the model from the individual and spreads it among the members of the team.

4.1.4 Review

The review is a technique similar in nature to the code inspection, but which is intended to give *management* and study sponsors evidence that the development process is being done according to stated system objectives [Hollocker 1987]. Its purpose is to evaluate the model in light of development standards, guidelines, and specifications. As such, the review is a higher level technique more concerned with the design stages of the life cycle. Reviews are frequently termed as "design" reviews.

As opposed to walkthroughs and code inspections, which have more of a correctness determination flavor, reviews seek to ascertain tolerable levels of quality are being attained. The review team is more concerned with design deficiencies and deviations from stated development policy than it is with the intricate line-by-line details of the implementation. This does not imply that the review team is free from the responsibility of discovering technical flaws in the model, only that the review process is geared towards the early stages of the development cycle. The review is also intended to identify subjective aspects such as performance improvement and economic aspects. It would seek to indicate that the preliminary and detailed programmed model designs are sufficiently valid, well-designed, and effective representations of the real-world system. The formal review gives the modeler evidence that the programmed model conforms to proven quality standards.

The review is conducted in a similar fashion as the code inspection and walkthrough. Each review team member examines the model prior to the review. The team then meets to evaluate the model relative to specifications and standards, recording defects and deficiencies. Ould and Unwin [1986] provide a design review checklist depicting some of the critical points to look for in a design. The result of the review is a document portraying the events of the meeting, deficiencies identified, issues resolved by management, and review team recommendations [Hollocker 1987]. Appropriate action may then be taken to correct any deficiencies.

4.1.5 Audit

The audit seeks to determine through investigation the adequacy of the overall development process with respect to established practices, standards, and guidelines. The audit also seeks to establish traceability within the development process. Given an error in a part of the model, the error should be traceable to its source in the specification via its audit trail. The audit verifies that model evolution is proceeding logically and that it is evolving in accordance with stated requirements [Bryan and Siegel 1987]. In doing so it gives visibility to the sponsor of what is being built, it provides a basis for communication among study participants, and it helps the modeler assess the scope of the study. This last item is particularly useful in helping the modeler avoid the Type III error, i.e., the error of solving the wrong problem.

Hollocker [1987] contrasts the audit and the review. The audit is accomplished through a mixture of meetings, observations, and examinations. It is performed by a single *auditor*. Auditing can consist of other audits, reviews, and even some testing, and it is carried out on a periodic basis.

4.1.6 Advantages and Disadvantages of Informal Analysis

Informal analysis can be of great importance to PMV. Its techniques are valuable from the early stages of Model Formulation throughout the entire programming process. In particular is the ability of informal analysis techniques to evaluate the subjective and multifaceted aspects of the simulation study. The success of a simulation study stems from the ability to achieve sufficiently correct simulation results and as importantly, to convince the study sponsor that the simulation model is a sufficiently valid one. Insuring the acceptance of the many subjective aspects of the model cannot be overlooked.

Besides the advantage of allowing human reasoning in the verification process, informal analysis techniques are not difficult to perform and require virtually no computer resources. On the other hand, the techniques used are very time consuming and require very high human resource allocation. Because of their reliance on human evaluation they are prone to human error. Success depends on the level of knowledge and expertise of the individual. The human time and effort required coupled with the likelihood of error result in limited effectiveness of informal analysis. Though their effectiveness improves as their guidelines for use become more structured and formal, informal analysis techniques cannot be relied upon in themselves to verify the programmed model.

4.2 Static Analysis

Static analysis is concerned with verification on the basis of characteristics of the static model source code. Static analysis does not require execution of the model. Its techniques are very popular and widely used, with many automated tools available to assist the analysis. The language compiler is itself a static analysis tool. Static analysis can be performed throughout the entire programmed model development process.

Static analysis techniques can obtain a variety of information about the structure of the model, coding techniques and practices employed, data and control flow within the model, syntactical accuracy, and internal as well as global consistency and completeness of implementation. The information gathered can be used to generate test data for use with other types of analysis, can identify the testing requirements for the various areas of the model, can be used to optimize the model's code, and can even be used to instrument the model to enhance further analysis. Just as importantly, static analysis results provide an indication of the principles used to meet the objectives of the study's software development project [Arthur et al. 1986].

Knowing that the model is being engineered for quality makes a strong statement for its verification.

Static analysis techniques vary in their degree of formality, ranging from informal to formal. For instance, checking consistency among submodel interfaces would not be considered as mathematically formal as would certain techniques for performing model data flow analysis [Allen and Cocke 1976]. Static analysis is generally more complex than informal analysis but not as complex as the other categories of analysis. The following sections explore the verification capabilities of static analysis techniques.

4.2.1 Syntax Analysis

Any model that is to undergo translation from a higher form to a machine-readable form must first pass a syntax check. This check assures that the mechanics of the language are being applied correctly. This fundamental analysis of the source code is by far the most widely utilized verification technique. It is unfortunate that most often this verification tool is utilized in the minimal way—getting the source code to successfully compile.

During the course of a compilation, as the syntax is checked and the source statements "tokenized," a symbol table is built which describes in detail the elements, or symbols, which are being manipulated in the model. This includes descriptions of all function declarations, type and variable declarations, scoping relationships, interfaces, dependencies, and so on. The symbol table is the "glue" which holds the compilation together, growing dynamically as the source code is scanned. Obviously there is a wealth of information about the static model available in the symbol table. Just listing the table itself is a tremendous source of documentation.

In addition to the symbol table, cross-reference tables are easily generated which provide such information as called versus calling submodels, where each data element is declared, referenced and altered, duplicate data declarations (how often and where occurring), and unreferenced source code. Submodel interface tables reflect the actual interfaces of the caller and the called, particularly useful when using a compiler that does not perform strict type checking nor verify external calls. Also readily created are maps which relate the generated runtime code to the original source code. All of this information is useful for documentation purposes. It is even more useful as the underpinnings for debugging. Examples of these types of listings can be found in the appendixes.

Another useful feature is the ability to reformat the source listing on the basis of its syntax and semantics. This enforces a level of uniformity among all coded submodels, which in turn promotes source code readability and ease of interpretation. Source code formatters, often referred to as "pretty printers," provide standard listing, clean pagination, and source code enhancement, such as highlighting of data elements (e.g., global variables, parameter variables, etc.) and marking of nested control structures. The formatted listing of the simulation case study from the previous chapter can be found in Appendix A.

All of the above have obvious merits for documentation and display of the source model, and even the model specifications. Fairley [1975,1976,1977,1978] extended the use of this information to other areas of analysis as well. He suggested capturing the analysis history in a data base and using it to drive and support other aspects of the verification process. A practical application of this idea is inserting probes into the source to enhance testing (see the section on Instrumentation in Chapter 1). The static data gives information about optimal placement of probes. Another example is the use of the symbol table and map to facilitate symbolic debugging, i.e., debugging at the source code level. Just as important, collected static data, later combined with model execution data, provides a powerful mechanism for verifying execution results, as was illustrated in the case study.

4.2.2 Semantic Analysis

Also occurring during source code translation is semantic analysis. Semantic analysis attempts to determine the modeler's intent in writing the code. The goal is to obtain an accurate translation of modeler intentions. In truth, the only meaning which can be derived from the source code is that which is self-evident in the code. It is dangerous to let the compiler make any other assumptions about modeler intentions. It therefore becomes beneficial, even to the point of being essential, to tell the modeler what it is that he has said in the source code (i.e., what his *code* means). The same principle can be applied to specifications. It is then up to the modeler to verify that the true intent is being reflected.

When the source code is being parsed during compilation, the target runtime system is most likely being simulated. This allows the compiler to generate code which will perform the requested tasks. As the meaning of the source code is derived, the corresponding runtime code is produced. The symbol table is referenced to check that the data elements used fit the operation being performed. A result of this inherent knowledge mechanism is the ability to determine what is and is not being used, how often it is being used, and to a large degree in what manner it is being used. As in syntax analysis, the harnessing of this information provides a healthy source of documentation.

Other benefits include locating variables which have been used but not initialized. This common model programming error can be the source of great frustration. Another common source of problems that can be identified is function side-effects, i.e., the actions of one operation intentionally or unintentionally altering the value of a supporting data item. This can be detected by noting when and where a variable gets changed. If a particular variable or code segment never gets used, chances are good that this is a symptom of some deeper problem. An example might be a constant conditional expression, or a variable that gets declared, maybe even initialized, but never used again. Even if there is no design error, space

is being wasted and the situation will inevitably lead to later confusion. This "dead code" is a prime target for optimization techniques which improve the performance and quality of the model. Examples of this information can be found in the appendixes. The Warning Report in Appendix E shows identifiers which are never used. Two identifiers declared in EXPON were never used. The Side Effects Report in Appendix G shows that several identifiers were referenced (*ref*) outside of their immediate scope, but were never altered.

It is probably worth noting here that neither syntax analysis nor semantic analysis require complete compilation in order to obtain their results. Most static analyzer tools simply apply the necessary steps to extract the data, without attempting to translate the code. Some of the algorithms required to accomplish some of these tasks can be rather complex (e.g., see [Allen and Cocke 1976]).

Like the results of syntax analysis, semantic analysis results should be captured and maintained to drive other parts of the verification process. The usefulness of this data will become self-evident as dynamic analysis techniques are discussed later.

4.2.3 Structural Analysis

Structured design and development refers to the use of widely accepted techniques for constructing quality software. These techniques are all founded on a set of principles which are recognized to be effective and comprehensive building blocks for software development. The principles are based on the use of acceptable "control structures" from which the software will be built. The three basic control structures are sequence, selection, and iteration. Appendix K contains flowcharts illustrating these structures. A well-built, structured programmed model can be decomposed into smaller and smaller submodels, each of which will exhibit these fundamental structured characteristics.

Structural analysis examines the model's structure and determines if it adheres to structured principles. This is accomplished by constructing a graph of the model control structure. This graph defines model control flow and as such is called a control flow graph. Figure 16 shows the control flow graph for the main routine of the programmed model in the previous chapter. It is considered to be structured. Figure 17 shows the control flow graph for an unstructured model segment. This graph is unstructured because it cannot be decomposed into the basic control structures. The control flow graph is analyzed for anomalies, such as multiple entry and exit points, excessive levels of nesting within a structure, and questionable practices such as the use of unconditional branches (i.e., GOTOs). The anomalies can be flagged so that they may be scrutinized further. Many of today's high-level languages are, by nature, structured. These structured languages not only encourage the use of structured programming techniques, they increase the ability to perform structural analysis. Structural analysis may also reveal commonalities of particular model structures. Steps may be taken to reduce the structure if possible. Figure 18 illustrates the reduction of the control structure for submodel PROCESS_EVENT.

The control flow graph is an effective verification document. It documents the model's control flow in a clear and concise way. A well-structured model naturally has a "clean-looking" control flow graph. A "clean" graph not only indicates a sound structure, it is easily understood and readily accepted even by the layman. It is a graphic illustration of the saying, "a picture is worth a thousand words."

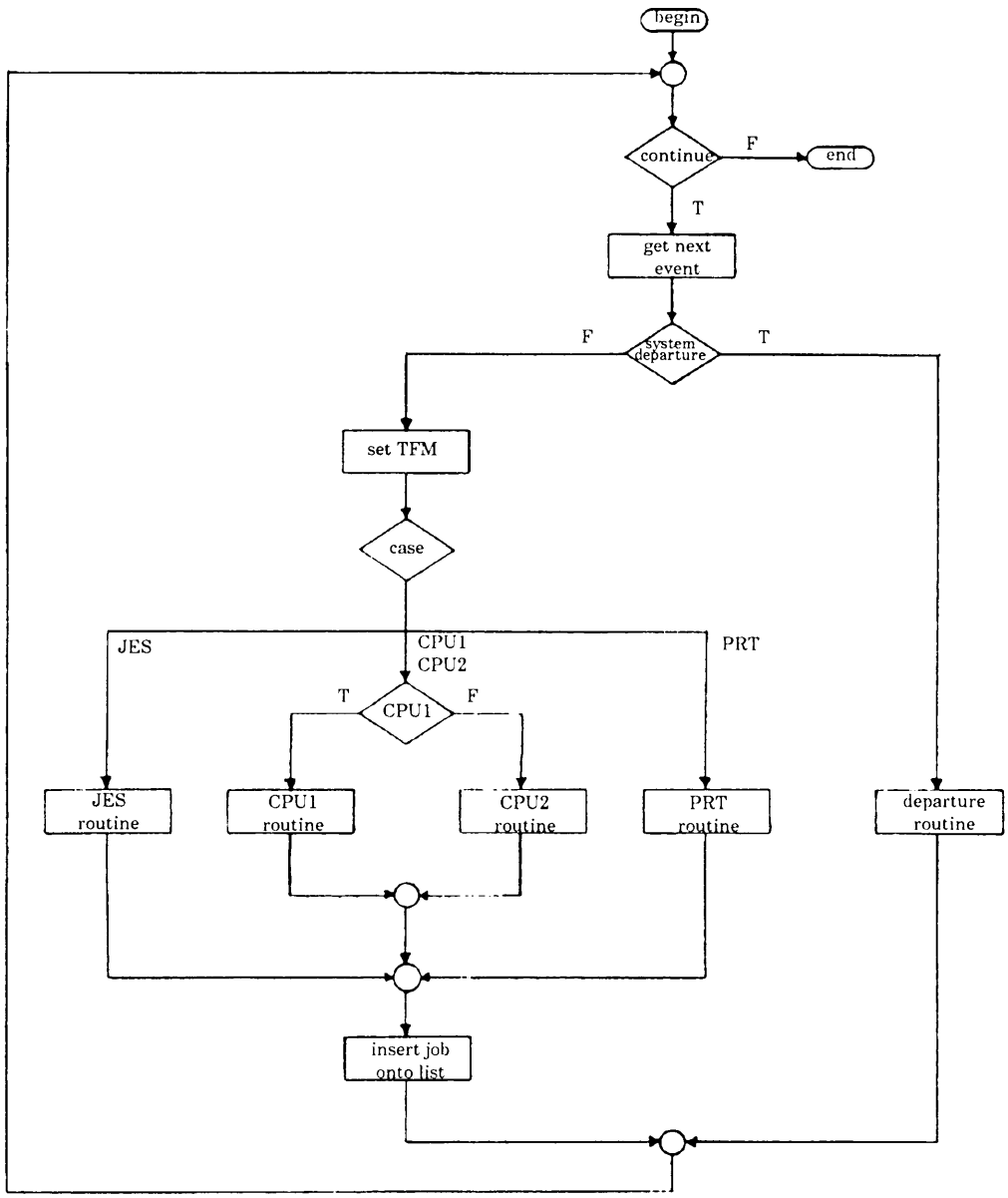


Figure 16. Structured Control Flow Graph

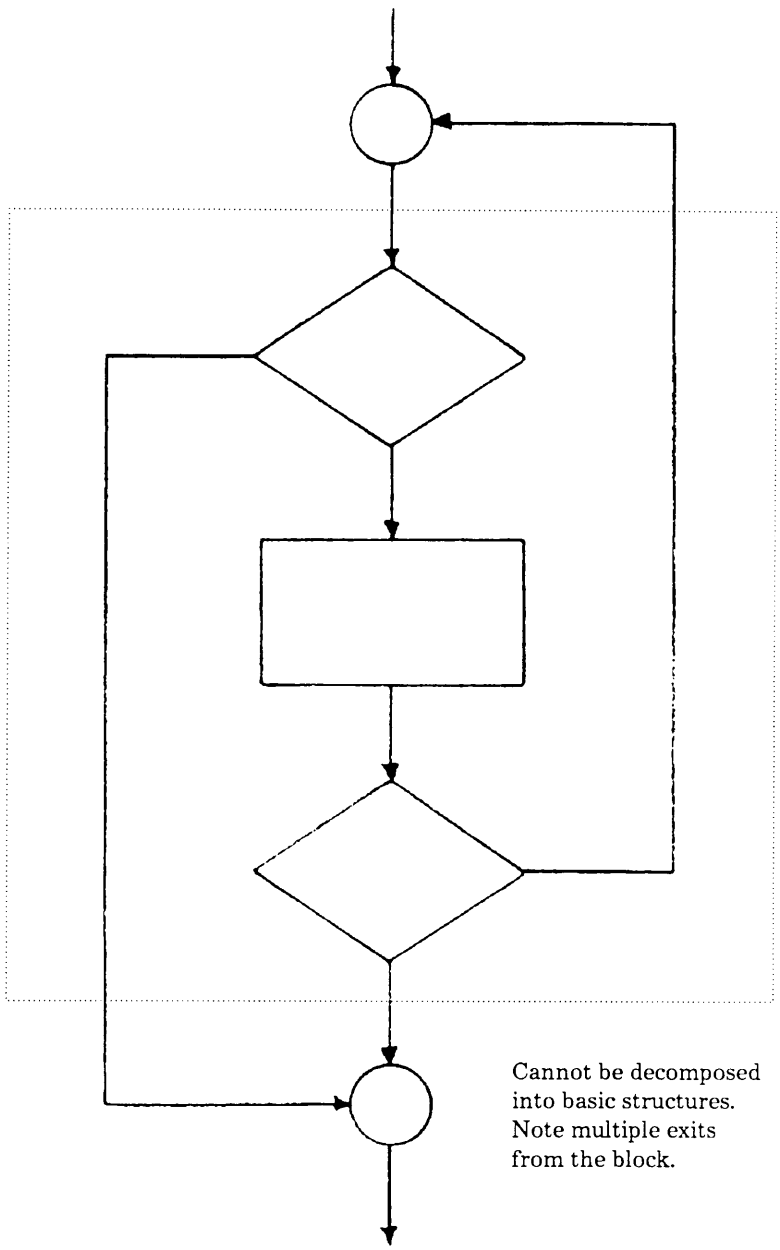
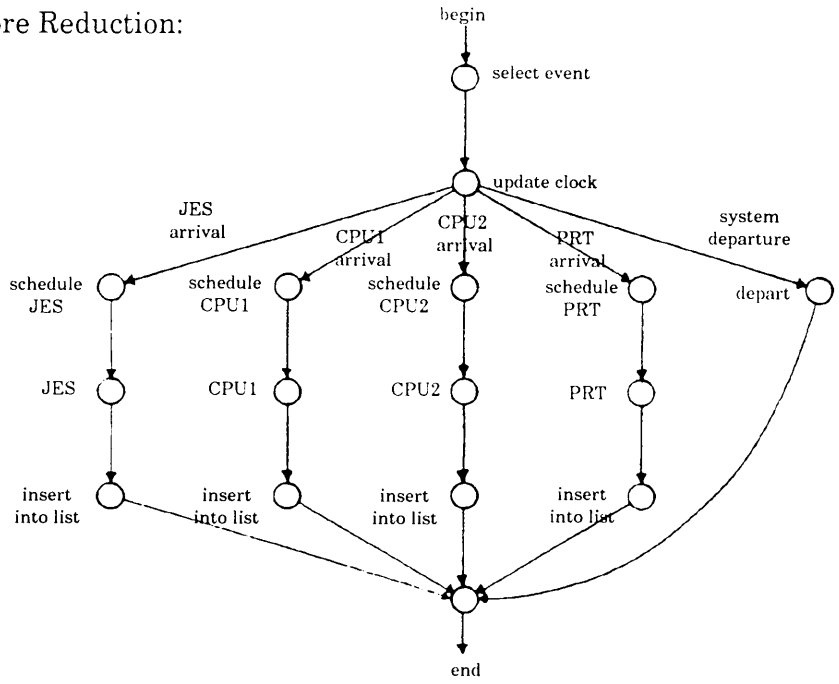


Figure 17. Unstructured Control Flow Graph

Before Reduction:



As Reduced:

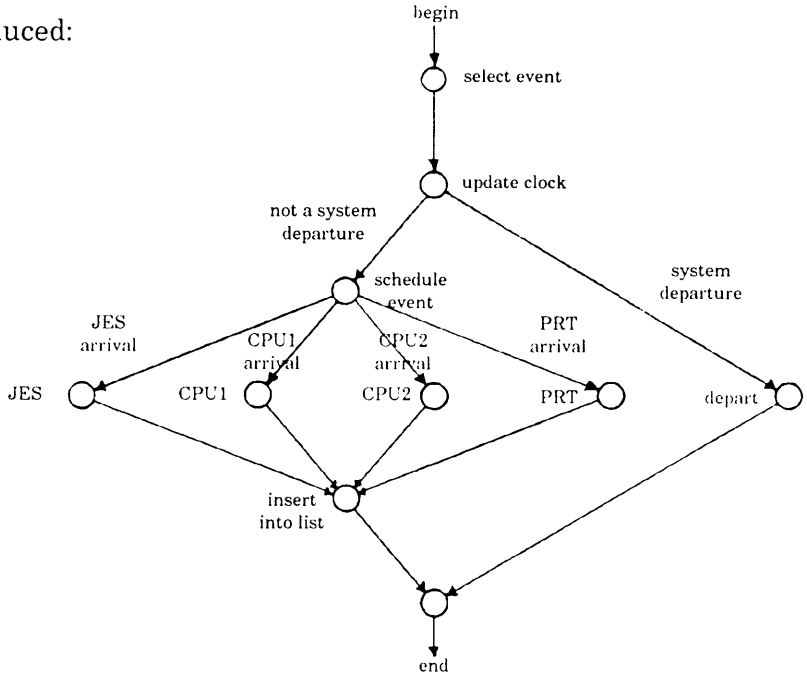


Figure 18. Reducing Model Structure via a Control Flow Graph

4.2.4 Data Flow Analysis

Data flow analysis is concerned with the behavior of the programmed model with respect to its use of model variables. This behavior is classified according to the definition, referencing, and unreferencing of variables [Adrion et al. 1982], i.e., when variable space is allocated, accessed, and deallocated. A data flow graph can be constructed to aid in the data flow analysis (see Figure 19). The nodes of the graph represent statements and corresponding variables. The edges represent control flow.

Data flow analysis can be used to detect undefined or unreferenced variables (much as in static analysis) and, when aided by model instrumentation, can track minimum and maximum variable values, data dependencies, and data transformations during model execution. It is also useful in detecting inconsistencies in data structure declaration and improper linkages among submodels [Ramamoorthy and Ho 1977].

4.2.5 Consistency Checking

Consistency checking is essential to the integrity of the model. It is intended, as Saib et al. [1977] put it, to prevent "apples being assigned to oranges." Consistency checking is concerned with verifying that the model description does not contain contradictions. All specifications must be clear and unambiguous so that each person viewing the model sees the same thing. All model components must fit together properly. Consistency checking is also concerned with verifying that the data elements are being manipulated properly. This includes data assignment to variables, data use within computations, data passing among submodels, and even data representation and use during model input and output (e.g., input prompts and output descriptions accurately reflect the meaning and use of the data). Much of consistency

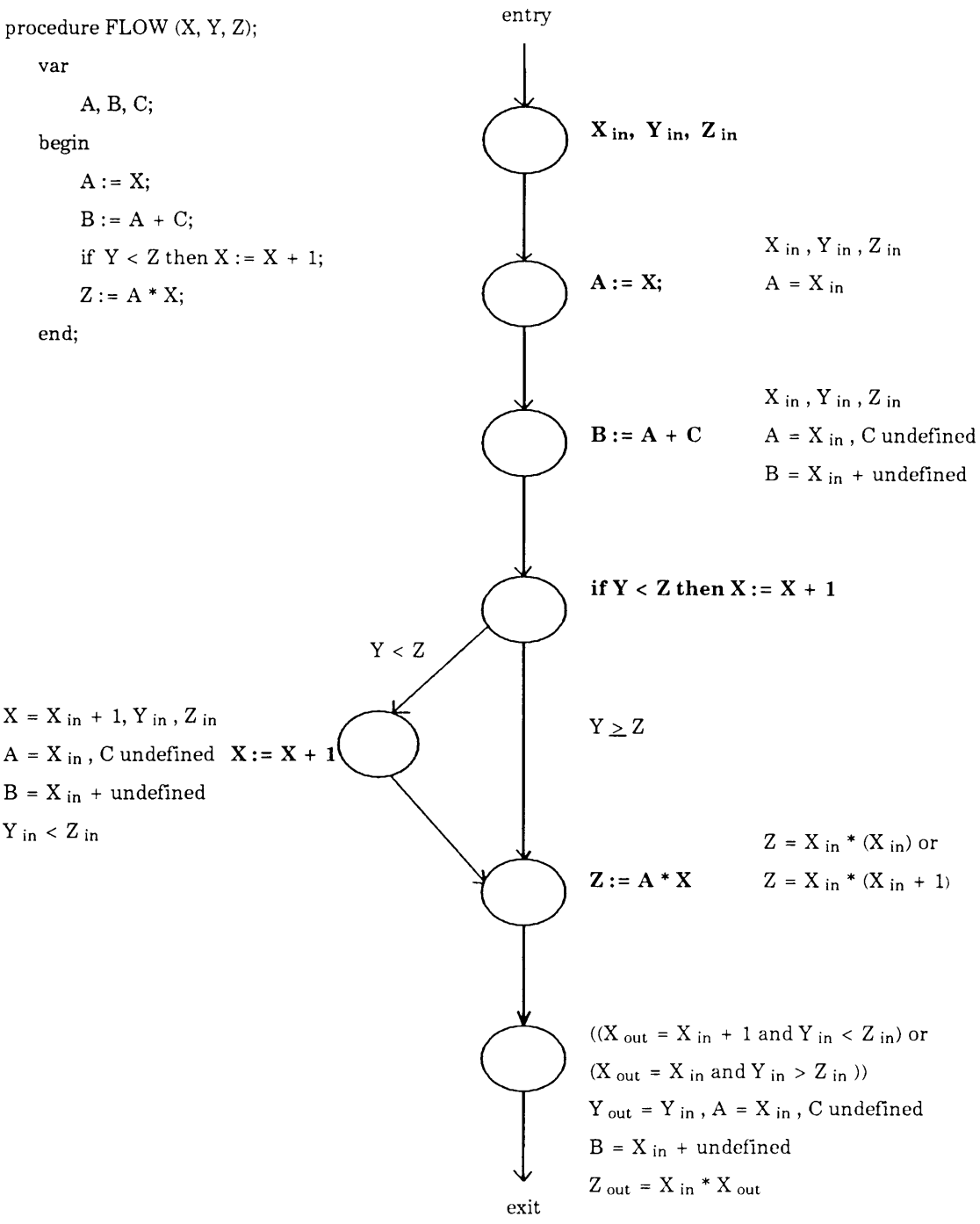


Figure 19. A Data Flow Graph

checking is accomplished by using the documentation produced by syntax and semantic analysis (listings, cross-references, etc.) as material to guide code inspections and walkthroughs. As the specification becomes more formally stated, more of the work can be automated. Data elements and interfaces can be checked as they are actually used to ensure their consistent usage.

All studies should maintain as part of their specification and documentation a data dictionary. The data dictionary defines the purpose and composition of each data item. By having the data dictionary on-line in a data base during development, consistency checking can be greatly enhanced. Language sensitive editors can query the dictionary each time a data element is declared or used, verifying that conflicts do not occur. Additionally, the data dictionary serves as a cross-reference source during compilation and similar analysis, and further aids subsequent phases of PMV.

Yet another perspective on consistency checking pertains to the cosmetic style with which language elements are applied (e.g., naming conventions, use of upper, lower, and mixed case, etc.). This perspective follows the same reasoning behind the creation of formatted listings with "pretty printers": cleaner presentation leads to ease of understanding. While seemingly a matter of taste with little merit for attention, cosmetic consistency has a significant standardization effect. From standardization follows better understanding, from better understanding, improved likelihood of added quality. A cosmetically consistent model can reduce the likelihood of Type I error.

4.2.6 Advantages and Disadvantages of Static Analysis

Most static analysis techniques have automated tools which support their use. As a result, the human resource cost is appreciably low. Since model execution is not involved, computer

resource cost is moderate compared to instrumentation-based verification approaches. These techniques are limited, however, in what they can actually verify. For instance, static analysis can verify that the syntax used conforms to the defined syntax of the language. It can make conclusions about the semantics of the model and inferences on aspects of the model's execution. It cannot insure that the intentions of the modeler are being met nor can it algorithmically examine a model to determine its execution behavior [Fairley 1978; Hopcroft and Ullman 1969]. Further, the basis for performing the verification must be shown to be correct (e.g., the compiler must be correct).

Overall, static analysis has proven to be an effective verification method. Its strength lies in the number of well-known techniques which are supported by a variety of commercially available tools, most of which are highly automated. Further, static analysis complements other methods of verification, such as symbolic execution and execution profiling, to name a couple.

Especially important to the simulation study is the extensive documentation generated through static analysis. Graphs which depict the model's logic and data flow are easily understood even through the layman's eyes. A good example of this was illustrated in the previous chapter when comparing the event scheduling world view flowchart (an accepted standard) with the corresponding structure of the programmed model. This enhances the credibility assessment of the model by opening an avenue of communication between the modeler and the sponsor, through which the sponsor can provide feedback to the modeler. Likewise, the construction of the model can be shown to be structurally sound and free of any anomalies which might arouse questions about the model's integrity.

4.3 Dynamic Analysis

Verification by dynamic analysis is accomplished by evaluating the model during its execution. As the model is exercised, its behavior is observed and information about its execution gathered.

Testing and dynamic analysis are often considered one and the same. This is probably because they both relate to exercising the model. However, their relationship is not to be misunderstood. Dynamic analysis encompasses much more than model testing. There are a variety of other techniques which are concerned with model execution behavior. Symbolic debugging, execution tracing, and execution monitoring are also dynamic techniques. Model testing is, however, the broadest area of dynamic analysis and perhaps the most common means thought of for verifying execution behavior. What more natural way to check if a model behaves as desired than to watch it execute?

Dynamic analysis is the traditional verification approach used by software developers. Because of the proliferation of dynamic analysis techniques among developers, it is not surprising that as a group these techniques are the most popular and commonly used. Techniques range from the ad hoc to the carefully researched.

Effective dynamic analysis has a moderate to high level of complexity. One area of complexity is determining what to test and how to test it. This can be an ominous undertaking even for moderately sized models (5,000 to 10,000 lines of code). The sheer number of execution paths a model might take makes complete testing prohibitive, if not impossible. Deciding which testing approach to take often becomes a battle of trade-offs between time and effort versus level of coverage obtained. Fortunately, static analysis and symbolic analysis are helpful in determining the testing needs of the various areas of the model. Instrumentation is also

helpful in preparing the model for collecting execution information. Another complexity is interpreting the analysis results. Presenting the data in a meaningful way is just one aspect of the problem. Applying the evidence to the goal of verification is another.

The high computer resource needs of dynamic analysis should be obvious. The human resource cost may not be so obvious. Some dynamic analysis techniques require continuous monitoring and activity by the modeler. On-line debugging, for instance, requires a heavy investment in modeler time. On the other hand, generating an execution trace requires little human effort at all. All dynamic analysis techniques require time to analyze the execution results. Human resource cost may become expensive. Like static analysis, most dynamic techniques are automated, with many well-known tools and techniques for performing it available. By allowing the observation of model behavior, dynamic analysis provides a good basis for verifying functional correctness.

Discussion of dynamic analysis techniques follows in the sections below.

4.3.1 Top-down Testing

As mentioned earlier, model testing is the broadest area of dynamic analysis. To be effective, there needs to be a well-disciplined plan for applying testing. Most models' testing needs are simply too immense to approach testing in a haphazard manner. Myers [1979] uses a simple problem to clearly illustrate how testing quickly mushrooms into an enormous task. Skeptics are advised to try this self-assessment test!

In a typical simulation study, the model will consist of several large submodels (or modules), each of which may operate on a separate processor. Each of these submodels may contain more submodels (or units). For the model to become operational, all of these model components must be integrated together. In addition to testing the individual models and submodels,

the integration of the model must be tested. This is known as integration testing. There are several approaches to testing. Some approaches are directional, proceeding from one level of the model to another. Other approaches are concerned with a particular view of the model, looking at what it produces or the details of how it was built. In practice, multiple approaches are blended to achieve comprehensive testing. Specific model designs often lend themselves to a particular approach. Thus, there is no correct approach. It is up to the modeler to decide which testing approach best fits a given situation. In this section, *top-down testing* is discussed.

To best understand top-down testing, one must discuss top-down model development. In top-down development, the modeler defines a global picture of the model which he then breaks into submodels. For each submodel, the process is repeated. When the model has been designed, implementation begins at the global (top) level of the model. When that level has been developed, the modeler similarly develops each submodel, until the model development is complete. Top-down development of programmed model SIMULATION (from the case study in Chapter 3) would involve implementing the main model driver first, followed by submodels WRITE_HEADING, WRITE_RESULTS, INITIALIZE, PROCESS_EVENT, and ARRIVAL, all of which are called from the global model level. In similar fashion, the remaining submodels would be implemented.

Top-down testing follows the same pattern as top-down development (although the two need not parallel each other). Top-down testing would begin with testing the global model and then proceed to testing the submodels. When testing a given level, calls to sublevels are simulated using submodel "stubs." A stub is a dummy model which has no other function that to let its caller complete the call. An example stub for PROCESS_EVENT is shown in Figure 20. This particular stub simply increments the variable ***completed_jobs***, which allows the main level to test its ability to terminate the simulation. Fairley [1976] lists the following advantages of top-down testing:

```
procedure PROCESS_EVENT(var event_list : event_ptr;
                        var system_data : system_description);

    begin

        with system_data do

            completed_jobs := completed_jobs + 1

    end;  (* test stub for PROCESS_EVENT *)
```

Figure 20. Test Stub for PROCESS_EVENT

1. Model integration testing is minimized,
2. Early existence of a working model results,
3. Higher level interfaces are tested first,
4. A natural environment for testing lower levels is provided, and
5. Errors are localized to new submodels and interfaces.

Some of the disadvantages of top-down testing are

1. Thorough submodel testing is discouraged (the entire model must be executed to perform testing),
2. Testing can be expensive (since the whole model must be executed for each test),
3. Adequate input data is difficult to obtain (because of the complexity of the data paths and control predicates), and
4. Integration testing is hampered (again, because of the size and complexity induced by testing the whole model). [Fairley 1976; Panzl 1976]

The opposite approach to top-down testing is bottom-up testing. Bottom-up testing is the topic of the next section.

4.3.2 Bottom-up Testing

Bottom-up testing follows bottom-up implementation. In bottom-up implementation, the system is coded from the submodel level up. As each submodel is completed, it is thoroughly tested. When the submodels comprising a model have been coded and tested, the submodels are integrated and integration testing performed. This process is repeated until the complete model has been integrated and tested. The integration of completed submodels need not wait for all "same level" submodels to be completed. Submodel integration and testing can be, and often is, performed incrementally. With the bottom-up strategy, the model is constructed from supposedly correct components.

This strategy encourages extensive testing at the submodel level. Since most well-structured models consist of many submodels, there is much to be gained by bottom-up testing. The smaller the submodel and more limited its function, the easier and more complete its testing will be. Bottom-up testing is particularly attractive for testing distributed systems.

One of the major disadvantages of bottom-up testing is the need for individual submodel drivers to test the submodels. These drivers, more commonly called test harnesses, simulate the calling of the model and pass test data necessary to exercise the submodel. The task of developing harnesses for every submodel can be quite large. In addition, these harnesses may themselves contain errors. A test harness for AREA_J (in the case study) is shown in Figure 21.

Another disadvantage, as Panzl [1976] points out, stems from the fact that once testing rises above the lower level submodels, bottom-up testing faces the same cost and complexity issues as does top-down testing past the higher levels. In both strategies, exhaustive testing of the interior submodels to opposite-end submodels (e.g., in top-down testing, the lower level submodels) is costly and difficult—if not impossible.

Mixed testing is a compromise to the top-down and bottom-up strategies. Under this approach, bottom-up testing is performed on submodels that cannot be tested top-down with mere stubs. Examples of such submodels are I/O models and interrupt handlers. The predominant technique in mixed testing is the top-down strategy.

Regardless of whether the strategy is top-down or bottom-up, some sort of environment simulation overhead is inherent. To be effective, the testing strategy must be well-planned and implemented so that it checks as many situations as possible, evenly distributed throughout the model, with the least incurred cost.

```
program AREA_J_HARNESS;
        .
        .
        .
    procedure AREA_J( ... );
        ...
    end;  (* procedure AREA_J *)

    function RAND( ... );
        ...
    end;  (* function RAND *)

    function EXPON( ... );
        ...
    end;  (* function EXPON *)

    procedure INITIALIZE_TEST_DATA( ... );
        ...
    end;  (* test harness procedure INITIALIZE_TEST_DATA *)

    procedure DUMP_STATE( ... );
        ...
    end;  (* test harness procedure DUMP_STATE *)

    begin  (* program AREA_J_HARNESS *)

        INITIALIZE_TEST_DATA(system_data);
        for i := 1 to test_cases do begin
            DUMP_STATE(system_data);
            AREA_J(system_data);
            DUMP_STATE(system_data);
            with system_data do begin
                clock := clock + EXPON(test_mean,test_seed);
                if RAND(test_seed) < 0.5 then
                    current_jobs := current_jobs + 1
                else
                    current_jobs := current_jobs - 1;
            end;
        end;
    end.  (* program AREA_J_HARNESS *)
```

Figure 21. Test Harness for AREA_J

4.3.3 Black-box Testing

Black-box testing is concerned with *what* the model or submodel does, i.e., what its function is. Black-box testing, also called functional testing, views the model as a black box. The concern is not what is in the box; rather, what is produced by the box. Testing of the model is accomplished by feeding inputs to the model and verifying the corresponding outputs. The model specification is used to derive test data [Myers 1979; Howden 1980].

Recalling the specification for JES_arrival events from the previous chapter, black-box testing of the model's handling of a JES_arrival might proceed as follows. An event list consisting of a number of JES_arrival events (several from each user category) is created. PROCESS_EVENT is called once for each arrival event on the test list. Submodel RAND is appropriately "fixed" to return pre-determined values so that (1) the duration of each event's service by the JES is known, and (2) the arrival time of new jobs can be controlled. Before each call to PROCESS_EVENT, the state of the model is dumped (current number of jobs, JES utilization time, the events list, the attributes of each event in the list, etc.); likewise, the state is dumped upon return from the call. In this way, the behavior of the model can be verified on the basis of how it transformed inputs to outputs in accordance with the model specification.

It is virtually impossible to test all inputs to the model. Rather than verifying that the model produces the correct output for each input, the modeler is more interested in finding inputs that produce incorrect outputs. Determining if the test set is complete is the main drawback to black-box testing [Westley 1979]. Black-box testing is typically used at the global model level, when all of the submodels have been thoroughly tested with another approach.

4.3.4 White-box Testing

As opposed to black-box testing, which tests the function of a model, *white-box testing* tests the model based on its internal structure (how it was built). White-box testing uses data flow and control flow graphs to verify the logic and data representations of the model. The focus of testing here is breadth of coverage of model paths. As many execution paths as possible should be tested.

Consider, for example, submodel DEPARTURE from the case study in Chapter 3. From the graphs of DEPARTURE in Figure 22 we verify that the model is structured and determine that there are three control paths which must be tested. The following test data will allow each path to be executed:

1. *in_steady_state* = true
2. *in_steady_state* = false **and** *completed_jobs* = *transient_length*
3. *in_steady_state* = false **and** *completed_jobs* ≠ *transient_length*

(The graph on the right of Figure 22 is a symbolic execution tree. Symbolic execution of this submodel allows us to simplify the expression of the test cases. A symbolic execution of this model would reveal that *in_steady_state* is equivalent to *completed_jobs* > *transient_length*, allowing simplification of the above test cases. Symbolic execution is discussed in section 4.4.1.) Each path should be analyzed to verify its activity. From the symbolic execution tree, we see that several variables assume that appropriate initialization has been done. Verifying this submodel would include ensuring that such assumptions are valid.

White-box testing is the most common mode of testing. It is the only reliable means of detecting redundant code, faulty model structure, and special case errors [Westley 1979]. An effective test plan determines which approach best fits the varied needs of the model and applies them accordingly. In most cases, all approaches will be used in some way, blended together in a well-orchestrated, concerted manner.

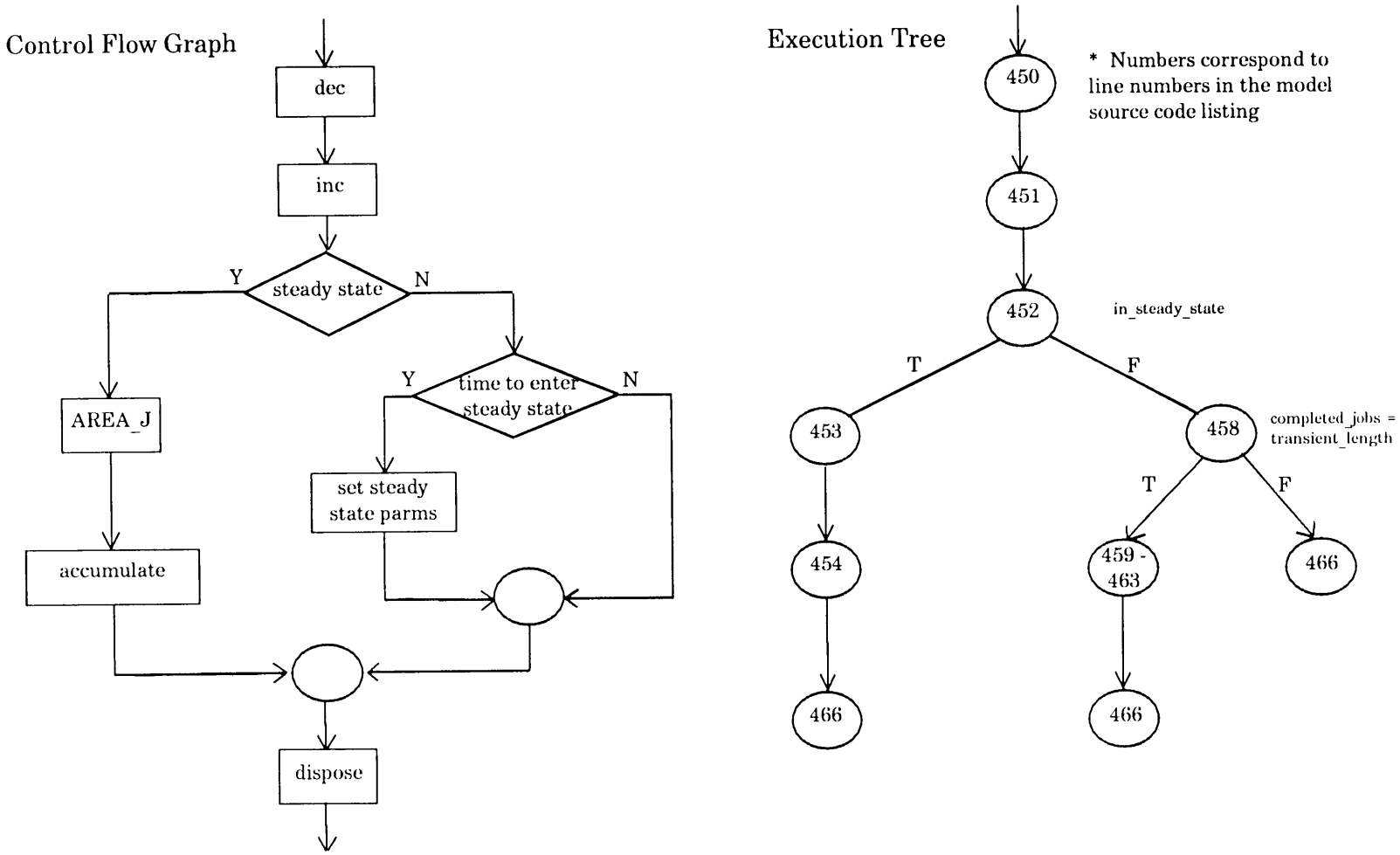


Figure 22. Control Flow Graph and Execution Tree for DEPARTURE

4.3.5 Stress Testing

A characteristic of simulation software is a dependency on time. Quite often real-time requirements and tight synchronization are involved. Testing these time-dependent situations is a difficult task. Many testing techniques are not adequate for these particular needs.

An approach to time-sensitive testing needs is stress testing. Stress testing is similar in nature to boundary analysis (see Section 4.5.3), with the critical parameter being time [Dunn 1987]. Stress testing tests the model on the borders of its time critical components. It pushes the model to and beyond its limits. As an example, consider a simulation model of a traffic intersection which specifies a maximum arrival rate of 50 cars per minute in a lane. A typical stress test would be a lengthy test forcing cars to arrive at or near the maximum arrival rate. In effect, the intersection becomes flooded with cars and the model's response in this situation can be monitored. Another test might be to exceed the maximum arrival rate for an extended period of time. If the model performs well under both valid and invalid input conditions, the model is said to be robust [Balci 1987]. As Myers [1979] points out, such tests are valuable because (1) such "never-will-occur" situations may, in reality, occur, and (2) system response under such conditions is often indicative of errors that might occur under "normal", less stressful conditions.

Stress testing, while in no way considered an exhaustive testing technique, is valuable for giving evidence (along the lines of strength in numbers) that a model will behave as desired if, after numerous stressful tests have been performed, no errors arise. Lack of errors do not imply correctness; however, stress testing provides an alternative to not having any functional evidence at all. It is important that any test plan involving stress testing be strongly supported with a solid structural testing program.

4.3.6 Debugging

Debugging is often confused with testing, much as testing is confused with verification. Testing reveals the presence of errors, debugging finds them and removes them. Debugging is an expensive technique. As [Dunn 1987] points out, 10 minutes of testing can result in 10 hours of debugging. Every effort should be made to remove defects before coding ever begins. Debugging, however, is an inevitable step of the simulation model development life cycle.

Given that errors have been detected by testing, debugging involves locating the source of error, determining the needs for correcting the error, making the correction, and then retesting the model to ensure successful modification. Probably the most difficult one of these tasks is isolating the true source of the error. Frequently, what may appear to be the source of the error is but an extension of a deeper problem. If the true source is not found, not only does the model remain incorrect, proposed "solutions" may in fact introduce other problems. The following sections discuss techniques which make debugging more effective.

4.3.7 Execution Tracing

Often times one of the best means of locating model defects is by *"watching"* the line-by-line execution activity of the model. This technique is known as execution tracing. Tracing is a very powerful means of verifying a model. The modeler can view the model's execution, determine what factors cause the traversal of particular paths, follow model data flow, determine in what order data elements combine and how the data is treated, and so on. Tracing is like creating a window into the execution environment. The modeler can see what is happening at specific locations in the model, recreate the events of the simulation, and easily track the source of errors.

Execution tracing is most often associated with interpretive languages. Interpretive languages offer source level tracing by simply displaying the source statement being interpreted at the given moment. Quite often development will be done using an interpretive version of the source language, then converting to a compiled version when development is complete. The tracing features and closeness to the source code of interpretive languages make this an attractive alternative. In compiled languages, tracing can be facilitated via model instrumentation (see Section 1.1.2 on instrumentation).

An execution trace can become very large very quickly. For this reason virtually all languages with any trace capability provide a mechanism for turning tracing on and off. Some languages, either directly or through instrumentation, pre-processing, etc., have facilities for generating traces only when certain exceptions occur, when certain model states are realized, or at specified points in the model code. Trace data can be displayed during execution or routed elsewhere for subsequent analysis and use. Fairley [1975,1976] suggests maintaining the trace data in a data base in order to enhance further verification activity.

Although execution tracing can be used to verify the model, other techniques are often easier to use, with the same or greater effectiveness. Typically, tracing is used to aid debugging by isolating known errors in the code.

4.3.8 Execution Monitoring

As a model executes, it is useful to monitor execution activity. Like tracing, execution monitoring provides a description of what the model is doing during execution. However, instead of giving a line-by-line account, monitoring gives information about activities and events which took place during execution. Monitoring may provide information about how many times the

model accessed a section of storage, or how long it took to perform a certain task. It may tell how many times the model was preempted by another job or how many times a page fault occurred (e.g., CPU utilization and waiting time). Execution monitoring provides an added dimension of information about model activity than does execution tracing.

Monitoring is accomplished by first instrumenting the code (see Section 1.2 on Instrumentation) with statements or submodels to perform the monitoring activity. When the simulation begins these submodels act as a shell around the actual model, allowing it to execute as normal except as required to gather execution information. In this way, hardware interrupts and other activities can be intercepted and processed as needed before passing control to the model. Except for the degradation of performance, the activities of the monitor are transparent. In order to minimize the execution slowdown, the monitoring may be done in a statistical manner. Instead of capturing every detail of model execution, the monitor submodels may take a sample at fixed intervals (say 20,000 times a second). During the interrupt, a quick recording of model state is made. The greater the sample size, the more detailed and reliable the result will be—at the expense of model execution speed.

Simulation models frequently involve distributed systems or real-time systems. Suppose, for example, a chemical process being modeled uses a number of hardware devices which communicate with each other via a message passing scheme. Messages are sent to a central dispatch processor, which in turn forwards the message to the appropriate receiving device. A concern of the model might be what percentage of the dispatcher's time is spent sending to, and receiving from, the various devices. Because of its hardware sensitive nature, execution monitoring would be useful in verifying these activities. Of course, for this example to be truly effective, care must be taken to ensure that the activity of the monitor does not seriously alter the events of the simulation. The effective use of execution monitoring constitutes a balance between the level of information obtained and the cost of obtaining it.

4.3.9 Execution Profiling

Execution profiling is a technique similar to execution monitoring. Profiling, however, is not as concerned with low level details as monitoring might be. Rather, profiling constructs a model profile which views matters on a much higher plane. While a monitor might check the number of times a communication signal was received, a profile would determine how many times the source code procedure which handles incoming signals was executed. The profile gives its results directly in terms of the source definition. The monitor, on the other hand, is more likely to provide memory addresses and port designations which will then have to be mapped to their source level equivalent.

Profiling requires instrumentation of the model (see Section 1.1.2 on Instrumentation) to map the runtime code to the corresponding source statement. When execution takes place, the instrumented model counts the number of times designated lines of the source code were executed or how often variables were referenced. A good profiling tool will allow the modeler to specify what level of profiling should be done. Useful information might be the number of times a submodel was entered, (i.e., how many times it was called), the number of times each line in a submodel was encountered, or the number of times a set of variables was referenced (e.g., global variables). This information, coupled with the knowledge of the test data that generated it, can verify proper control flow and data access, as well as show where the model is spending its time and what improvements and/or corrections can/must be made.

As was demonstrated in the case study in Chapter 3, execution profiling is a very useful verification technique. In the simulation case study, the profile gave evidence that the model was entering its steady state at the proper time, and that the model terminated when it was supposed to. Although a profile gives evidence for that execution alone, the evidence will accumulate with each test case.

Perhaps surprising to some, execution profiling tends to be more costly than execution monitoring. This is because a count must be kept of each line or element designated. Each time a line is encountered, execution must be interrupted and the count incremented. Since the profile is intended to be an actual count, it cannot be aided with statistical methods to increase its performance. Further slowdown occurs when mapping activity to the source level. Like monitoring, effective use of profiling requires care and consideration.

4.3.10 Symbolic Debugging

Symbolic debugging is a technique which uses a debugging tool that allows the modeler to manipulate model execution while viewing the model at the source code level. By setting "breakpoints", the modeler can control the conditions under which he interacts with the model. He may want to interact with the entire model one step at a time, or, as is more commonly the case, at predecided locations or under specified conditions. When using a debugger, the modeler is not merely a spectator. He may alter model data values or cause a portion of the model to be "replayed", i.e., executed again under the same conditions (if possible). Typically, the modeler will utilize the information from execution history generation techniques, such as tracing, monitoring, and profiling, to isolate a problem or its proximity. He will then proceed with the debugger to understand how and why the error occurred.

The earliest debuggers operated at the machine level, or at best, the assembly level. Using the debugger meant hours of tedious perusal of core dumps and conversion of hexadecimal codes. Current state-of-the-art debuggers allow viewing the runtime code as it appears in the source listing, setting "watch" variables to monitor data flow, viewing complex data structures, and even communicating with asynchronous I/O channels. Figure 11 illustrates a symbolic debugging session. The use of symbolic debugging can greatly reduce the debugging effort

while increasing its effectiveness. Symbolic debugging allows the modeler to locate errors and check numerous circumstances which lead up to the errors.

4.3.11 Regression Testing

By definition, life cycle implies change. As model development progresses the model is going to evolve: evolve to incorporate design changes, evolve to correct mistakes. Verification is also a continuous process, flowing with the tide of change. It is imperative, however, that verification not get lost in this sea of change. PMV must be able to keep abreast of the ebbs and flows of development.

When mistakes are corrected, the corrections often result in adverse side-effects to the existing model. If care is not taken, the correction of an error in one place leads to an error in another. The later in the life cycle error correction takes place, the greater the likelihood of harmful side-effects occurring. Regression testing seeks to assure that model corrections do not initiate other problems. Regression testing is usually accomplished by retesting the corrected model with a subset of the previous test sets used. This makes retaining and managing old test data essential. Successful regression testing is as much a matter of planning and configuration control (simulation project library management, version control, traceability, etc.) as it is anything else. Thus a plan for performing regression testing much be incorporated in the overall model design. Waiting until the first (sub)models begin undergoing correction and revision is too late to think about regression testing.

4.3.12 Advantages and Disadvantages of Dynamic Analysis

Dynamic analysis is not without its limitations. As alluded to earlier, the potential cost in human resources can be high. If not managed properly, dynamic analysis can needlessly consume the time of the modeler. Secondly, dynamic analysis cannot show model correctness. It can only reflect how the model behaves for a given set of test data. The possible test sets for a model can be infinite. Thus *complete* testing is rendered impossible for virtually all practical models of any speakable size. *Adequate* test coverage is a problem as well. The required scope of coverage broadens in exponential fashion as the model increases in size. Dynamic analysis does not possess the capability to manage this situation.

On the other hand, dynamic analysis techniques thoroughly document a given test execution. It can provide conclusive proof that a model functioned as intended. Dynamically executing the model is the only way to test (or "see") how the model behaves on a given hardware, or when operating on distributed hardware. The execution history not only enhances error detection and correction, it serves as a reference of model structure which can be used to enhance and maintain the model. Combining dynamic analysis with other verification techniques helps reduce some of the problems associated with dynamic analysis.

4.4 Symbolic Analysis

As pointed out in the previous section, dynamic analysis' effectiveness is limited because of the inability to verify all possible test cases. There is an approach to verification, however, that directly addresses this particular problem.

Symbolic analysis is an approach to verification that provides symbolic inputs to a model and produces expressions for the output which are derived from the transformation of the symbolic data along model execution paths. The basis for the verification is the transformation of inputs to outputs during execution. Symbolic analysis, like dynamic analysis, seeks to determine the behavior of the model during execution. It is a formal way of determining cause and effect relationships within the model. Some symbolic analysis techniques verify classes of input test data while others reduce the verification needs through the generation of effective test data.

As is shown in Figure 5, symbolic analysis can be used in a number of CASs during the Simulation Model Development Life Cycle. Because of its ability to deal with abstractions [Howden 1977] symbolic analysis is an effective means of verifying specifications. Its usefulness during programming is self-evident.

The simulation model is constructed in accordance with certain assumptions about the system being modeled. After the model is built, it undergoes experimentation. If the assumptions of the model are violated during experimentation, the model becomes invalid, even though the programmed model may function in a seemingly normal manner. As will be discussed in more detail later, symbolic analysis, when used in conjunction with constraint analysis, is a powerful tool for verifying conformance with model assumptions.

4.4.1 Symbolic Execution

Symbolic execution is the primary means of performing symbolic analysis. It is performed by executing the model using symbolic values rather than actual data values for input. During execution, the symbolic values are transformed as defined by the model and the resulting expressions are output. For example, consider the simple code segment of *Procedure ONE* in Figure 23. The result of the symbolic execution of this code would be the expression

```
Procedure ONE(X);

    begin

        A := 5;
        B := 10 * A;
        X := X * A + B;

    end;
```

Figure 23. Procedure ONE

```
X = 5 * X + 50.
```

Symbolically executing *Procedure TWO* in Figure 24 yields the following expression:

```
if  2 * X * X - 8 < 0  then  X := 0
else  X := 1
```

which is equivalent to

```
if  -2 < X < 2  then  X := 0
else  X := 1.
```

The functionality for *all* test data for *Procedure TWO* can be seen in this expression.

When unresolved conditional branches are encountered, a decision must be made which path to traverse. Once a path is selected, execution continues down the new path. At some point in time, the execution evaluation will return to the branch point and the previously unselected branch will be traversed. All paths eventually are taken.

The result of the execution can be represented graphically as a symbolic execution tree [King 1976; Adrion et al. 1982]. The branches of the tree correspond to the paths of the model. Each node of the tree represents a decision point in the model and is labeled with the symbolic values of data at that juncture. The leaves of the tree are complete paths through the model

```
Procedure TWO(X);

    begin

        A := 2;
        B := A * X;
        C := 8;
        D := B * X - C;
        if D < 0 then X := 0
                 else X := 1;

    end;
```

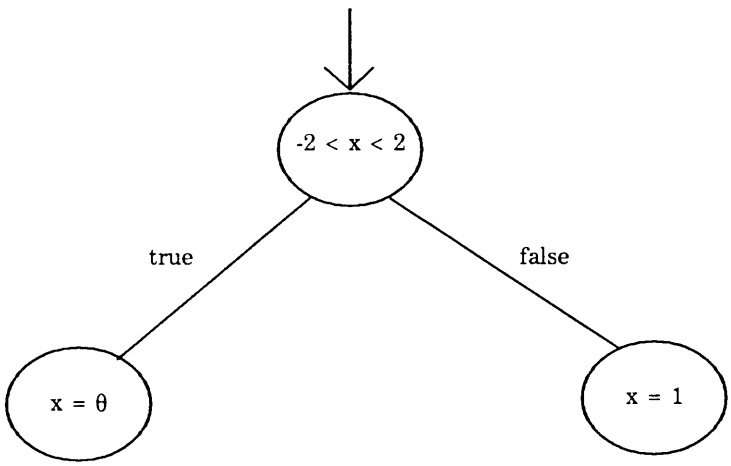


Figure 24. Procedure TWO and its Execution Tree

and depict the symbolic output produced. Figure 24 shows the execution tree for *Procedure TWO*.

As Westley [1979] points out, a big advantage of symbolic execution is in showing path correctness for all computations regardless of test data. One symbolic representation replaces a potentially infinite number of actual test cases. There are other advantages.

Consider the code of *Procedure THREE* in Figure 25. Through symbolic execution it can be determined that $C < 0$ will never occur and therefore the conditional expression is unnecessary. By symbolically representing model expressions, code segments in need of reconstruction can be identified and corrected [Howden 1977; Westley 1979].

Symbolic execution is also a great source of documentation [Osterweil 1983]. The resulting execution tree is in essence a symbolic trace of model function along its execution paths. Osterweil goes on to state, however, that the most important use of symbolic execution is as an aid to assertion checking, a type of constraint analysis. Constraint analysis verifies the model assumptions at critical points in the model (e.g., decision points) and symbolic execution verifies the behavior along the paths between constraint checks.

There are some problems with symbolic execution. Foremost is the issue of size. The execution tree explodes in size as the model grows. If the model is structured, then this problem can be relieved by analyzing subtrees of the model [Westley 1979]. This technique was utilized frequently during the case study verification in Chapter 3. Loops cause difficulties with symbolic execution. Since all paths must be traversed, loops make thorough execution impossible. This problem can usually be resolved by inductive reasoning, with the help of constraint analysis [Westley 1979; Adrion et al. 1982]. (When verifying *area* in the previous chapter, a type of inductive reasoning was applied when justifying the initial and subsequent values comprising *area*.) Symbolic execution is also limited in its use with complex data structures because of difficulties in symbolically representing particular data elements within

```
Procedure THREE(X,Y);

    begin

        A := 2;
        B := A * X;
        C := B * X;
        if C < 0 then Y := - C
                 else Y := C;

    end;
```

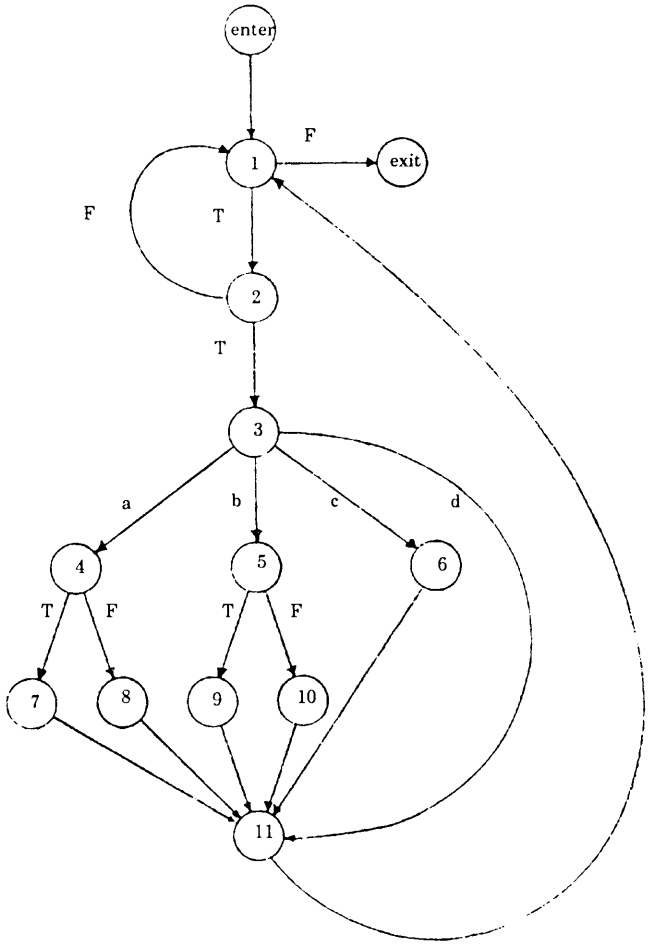
Figure 25. Procedure THREE

the structure [Hausen and Mullerburg 1983; King 1976; Ramamoorthy et al. 1976]. Since symbolic execution can be so difficult and cumbersome, its use is advocated only in systems with stringent reliability requirements [Ould and Unwin 1986]—much like a simulation model.

4.4.2 Path Analysis

The path analysis testing strategy [Howden 1976] attempts to verify model correctness on the basis of complete testing of all model paths. To perform path analysis, it is first necessary to determine the model's control structure (e.g., through structural analysis). This is followed by generating test data which will cause select model paths to be executed. Symbolic execution can be used to identify and group together classes of input data based on the symbolic representation of the model. The test data is chosen in such a way as to provide the most comprehensive path coverage possible. Among the coverage criteria sought are: (1) statement coverage, (2) node coverage (encounter all nodes), (3) branch coverage (cover all branches from a node), (4) multiple decision coverage (achieve all decision combinations at each branch point), and (5) path coverage (traverse all paths) [Prather and Myers 1987]. By selecting appropriate test data, the model can be forced to proceed through each path in its execution structure, thereby providing comprehensive testing.

In practice, only a subset of possible model paths are selected for testing. Recent work has sought to increase the amount of coverage per test case or to improve the effectiveness of the testing by selecting the most critical areas to test. Consider the flow graph in Figure 26 (an adaptation of submodel PROCESS_EVENT). By virtue of the looping structure, this model has an infinite number of paths. However, there are five decision nodes in the graph and 8 "basic" paths to be covered. Test data should be generated to cause each of these paths to be traversed. This set of test data is the test domain. The *path prefix* strategy [Prather and Myers 1987] is an "adaptive" strategy that uses previous paths tested as a guide in the selection of subsequent test paths. This strategy works as follows. From the test domain, select data to



Decision Nodes: 1, 2, 3, 4, 5

Basic Paths:

1 - exit
1 - 2 - 1 - exit
1 - 2 - 3 - 4 - 7 - 11 - 1 - exit
1 - 2 - 3 - 4 - 8 - 11 - 1 - exit
1 - 2 - 3 - 5 - 9 - 11 - 1 - exit
1 - 2 - 3 - 5 - 10 - 11 - 1 - exit
1 - 2 - 3 - 6 - 11 - 1 - exit
1 - 2 - 3 - 11 - 1 - exit

Figure 26. Path Analysis Example

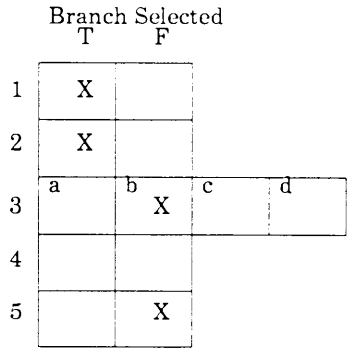
test a single path. Test that path and record the coverage obtained. Coverage can be recorded as in Figure 27. While paths remain uncovered, select the first untested branch of a previously tested branch, i.e., the shortest *path prefix*, and invert (or change) its result (e.g., change the value at node 1 from true to false, node 3 from JES to PRT, etc.). Select test data to cause the new path prefix to be executed, execute the path, and record the coverage. Prather and Myers [1987] prove that the path prefix strategy achieves total branch coverage.

The identification of *essential paths* [Chusho 1987] is a strategy which reduces the path coverage required by nearly 40 percent. The basis for the reduction is the elimination of non-essential paths. Paths which are overlapped by other paths are non-essential. The model control flow graph is transformed into a directed graph whose arcs (called primitive arcs) correspond to the essential paths of the model. Non-essential arcs are called inheritor arcs because they inherit information from the primitive arcs. The graph produced during the transformation is called an inheritor-reduced graph. Chusho presents algorithms for efficiently identifying non-essential paths and reducing the control graph into an inheritor-reduced graph, and for applying the concept of essential paths to the selection of effective test data.

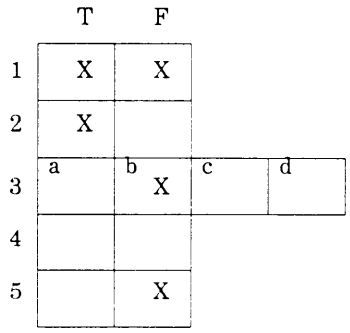
4.4.3 Cause-effect Graphing

Cause-effect graphing [Myers 1979] is a technique that aids in the testing of combinational input data by providing systematic selection of input condition subsets. Cause-effect graphing is performed by first identifying causes and effects stated in the model specification. Causes are input conditions, effects are transformations of output conditions. The causes and effects are listed, and the semantics are expressed in a cause-effect graph. The graph is annotated to describe special conditions or impossible situations. Once the cause-effect graph has been constructed, a limited-entry decision table is constructed by tracing back through the graph

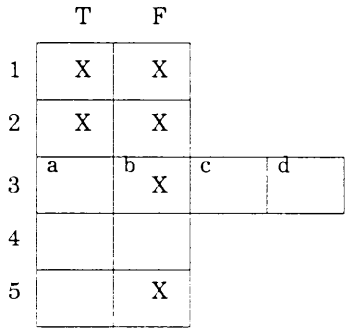
1. Select and apply a test case.
   Test case results in 1 - true
   2 - true
   3 - b
   5 - false
   Mark coverage with X.



2. Select and apply a test case which will invert 1 (i.e., results in 1 being false).
   Test case results in 1 - false
   Mark coverage with X.



3. Select and apply a test case which will invert 2 (i.e., result in 2 being false).
   Test case results in 1 - true
   2 - false
   Mark coverage with X.



4. Select and apply a test case which will alter 3 (e.g., change 3 to c).
   etc.

Figure 27. Recording Path Coverage

to determine combinations of causes which result in each effect. The decision table is then converted into test cases.

As an example, the following specification is given from the case study in Chapter 3. The enumerated causes and effects, the corresponding cause-effect graph, and the decision table are shown in Figure 28.

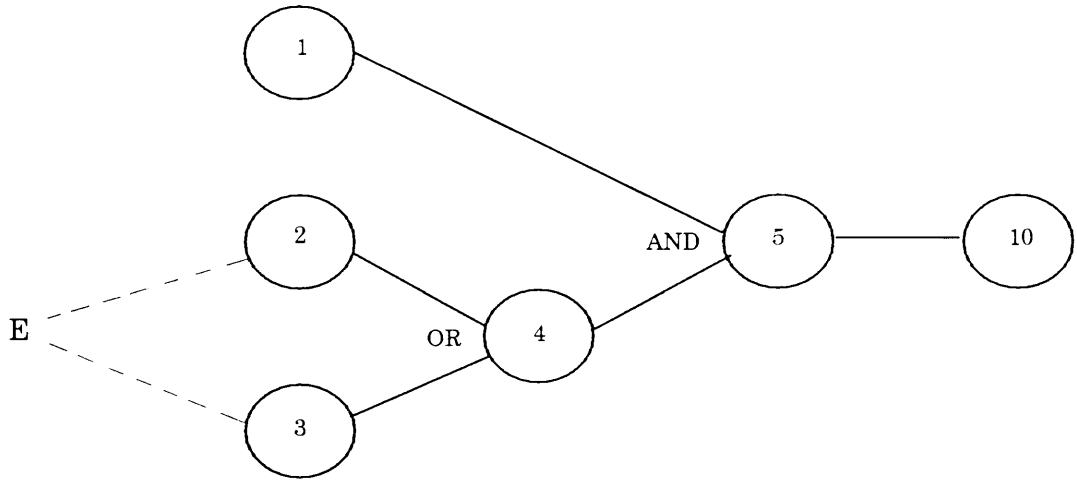
```
If the system is in the steady state, and
   either the event is an arrival or
   the event is a departure,
then the accumulated area will be computed.
```

The causes and effect are derived directly from the specification. The cause-effect graph is the equivalent of a combinational logic circuit (boolean) diagram. The dashed lines preceded by *E* annotate the exclusive-OR relation of causes 2 and 3. The columns of the decision table correspond to individual test cases while the rows reflect the state of the causes for each test case, or the resulting state of the effect.

A typical cause-effect graph and corresponding decision table will have numerous causes and effects. For this reason, the submodel must be dissected into segments small enough to be workable. This working size will be dependent on the nature of the model. The outcome of cause-effect graphing is a relatively small set of high-yield test cases, as well as a unique graphical description of the model. Myers [1979] provides a very detailed example of cause-effect graphing.

4.4.4 Partition Analysis

Partition analysis [Richardson and Clarke 1985] is a means of verifying the consistency of a model against its specification while at the same time generating comprehensive test data. It is, in a sense, a method of submodel testing. Partition analysis is accomplished by (1)



Cause

1	System is in the steady state
2	Current event is an arrival
3	Current event is a departure

Effect

10	*area* will be computed

Test Cases

		1	2	
	1	1	1	0 cause or effect absent
Causes	2	1	0	1 cause or effect present
	3	0	1	
Effect	10	1	1	

Figure 28. Cause-effect Graph

partitioning the model domain into submodels, (2) comparing the elements and prescribed functionality of each submodel specification with the elements and actual functionality of each submodel implementation, and (3) deriving test data which will extensively test the functional behavior of the submodel.

Partitioning is done by decomposing both specification and implementation into functional representatives. The decomposition is derived through the use of symbolic evaluation techniques, which maintain algebraic expressions of model elements and show model execution paths. Once partitioned, the functional representations are compared. These functional representations are the model computations. Two computations are equivalent if they are defined for the same subset of the input domain which causes a set of model paths to be executed, and if the result of the computations is the same for each element within the subset of the input domain [Howden 1976]. Standard proof techniques are used to show equivalence over a domain. When equivalence cannot be shown, partition testing is performed to locate errors—or, as Richardson and Clarke [1985] state, to increase confidence in the equality of the computations due to the lack of error manifestation. By involving both the specification and the implementation in the analysis, partition analysis is capable of providing more comprehensive test data coverage than other test data generation techniques.

4.4.5 Advantages and Disadvantages of Symbolic Analysis

In itself, symbolic analysis is an expensive method of verification. The generalizations of input data can be difficult to obtain and deriving the symbolic expressions can be an extremely complex task. Even if the symbolic expressions can be derived, their complexity may render them meaningless. Human resource cost can easily become unreasonably high, both in deriving symbolic results and in interpreting the results.

The effectiveness of symbolic analysis lies not in its standalone use, but as an auxiliary for other verification methods. Cause-effect mapping and partition analysis, for example, can generate effective test data for use with dynamic analysis. Symbolic execution can verify classes of test data, making dynamic analysis more effective in other areas of verification—areas where other methods may be less effective or less practical. The complementary relation of symbolic analysis and constraint analysis will be discussed in the following section.

4.5 Constraint Analysis

Early in the Simulation Model Development Life Cycle, when the model is formulated and specifications created, certain assumptions are made about the nature of the model. The simulation model operates within fixed boundaries. The model must completely contain the real-world system it represents [Balci and Nance 1985]. Otherwise Type III Error results. The Conceptual Model (preliminary model specification) details the constraints (boundaries, assumptions) on the model.

Constraint analysis verifies on the basis of comparisons between model assumptions and actual conditions arising during model execution. It additionally provides a level of validation. Constraint analysis has formal foundations, though not so formal as to be impractical to apply. Because of its formality, it has very powerful verification capability. Short of formal proof of correctness, constraint analysis provides the highest degree of PMV. Assertion checking, inductive assertions, and boundary analysis are the three techniques of Constraint Analysis which are discussed next.

4.5.1 Assertion Checking

Assertion checking verifies that the programmed model is performing according to its specification. It does this by comparing actual model state information with intended model behavior. Assertion checking accomplishes this by using assertions placed in the model to monitor model activity.

An *assertion* is a statement that should hold true as the programmed model executes [Balci 1987]. The purpose of an assertion is to check what *is* happening against what the modeler *assumes* is happening. Consider, for example, the following pseudo-code:

```
Base := Hrs * PayRate;
Gross := Base * (1 + BonusRate);
```

In just these two simple statements, many assumptions are being made. It is assumed that *Hrs* is non-negative; the same is assumed for *PayRate* and *BonusRate*. If the assumption is not true, *Gross* is meaningless, or even worse for some innocent employee, disastrous! Asserted code for this same segment might look like:

```
Assert(Base ≥ 0 and PayRate ≥ 0 and BonusRate ≥ 0);
Base := Hrs * PayRate;
Gross := Base * (1 + BonusRate);
```

Clearly, the assertion serves two important needs. First, the assertion statement verifies that the model is functioning within its given domain. Secondly, the presence of the assertion statement documents the intentions of the modeler.

Assertion statements which have been placed into the model's code as part of the runtime model are called *dynamic* or *executable* assertions. Placing assertions into the code is a form of instrumentation (see Section 1.1.2 on Instrumentation). This type of instrumentation is not likely to be automated. Placement of assertions requires deliberation on the part of the

modeler. The more formal the model specification, the easier this task will be. Symbolic analysis is helpful in determining effective placement of assertions. Symbolic analysis results in a graphical representation of model control flow, making it easier to locate effective places to put the assertions.

Assertion statements are typically entered into the source code as some form of comment. Most languages do not provide assertion features. Some languages do, however, allow assertions to be placed as comments in the source code and, through the use of a preprocessor, generate runtime assertion checking code. The ideal situation is to have a language which includes an assertion statement feature which can be activated at runtime as desired.

The idea of assertion features within a language dates back as far as 1972 when Satterthwaite [1972] included an ASSERT statement in his version of Algol W. Other authors have published similar work [Andrews and Benson 1979; Chow 1976; Fairley 1975; Hetzel 1973; Stucki and Foshee 1975]. Taylor [1980] acknowledges the shortcomings of most languages in accommodating assertion checking and provides a summary of suggested and recommended assertion features. These features include such things as defining the scope of an assertion's activity (locally, regionally, or globally; see [Balci 1987]), the ability to quantify assertions (*forall* and *exists* operators), the ability to reference previous variable values, and the ability to control the support environment (which assertions are active, what actions are taken on violation, features to control overhead, etc.). Even if a language does not include such features, Taylor's list provides guidance for developing procedures to be embedded into the code to perform the necessary assertion checking. The pseudo-code example above is written in such a way as to suggest the presence of an *Assert* procedure which is passed the result of the conditional expression and can then take appropriate action. Stucki [1977] provides a thorough discussion on the suggested use of assertions. Andrews and Benson [1979] extend the discussion to include the operators of first-order predicate calculus (implications, existence, and

universal quantifiers mentioned above). Balci [1987] demonstrates the use of assertion checking in GPSS/H code.

Assertion checking is expensive to implement. Expense comes in both human and computer resource cost. It is, however, a powerful verification technique. It provides the modeler a means to verify conformance to model specifications. It also provides documentation of modeler's intentions within the source code. When combined with symbolic analysis techniques such as symbolic execution, assertion checking becomes a very comprehensive means of verification. Assertions at the entry and exit points of a submodel verify the transformation of input to output states. Symbolic analysis can be used to verify what takes place between the assertions.

4.5.2 Inductive Assertions

The use of inductive assertions [Floyd 1967; Knuth 1968; Hoare 1969; Manna et al. 1973; Reynolds and Yeh 1976] provides the most "formal" constraint analysis verification and is, in fact, very close to formal proof of model correctness. This method requires the modeler to write input-to-output relations for all model variables. These relations are then written as assertion statements and placed into the model along model paths. The assertions are placed along the paths in such a way as to divide the model into a finite number of "assertion-bound" paths, i.e., an assertion statement lies at the beginning and end of each model path. The number of paths is made finite by placing an assertion within each loop in the model. These paths correspond to the compile-time traversal of the model rather than the run-time traversal [London 1977]. Verification is achieved by proving that for each path, if the assertion at the beginning of the path is true, and all statements along the path are executed, then the assertion at the end of the path is true. If all paths plus model termination can be proved, by induction, the model is proved to be correct.

4.5.3 Boundary Analysis

A model's input domain can usually be divided into classes of input data (known as equivalence classes) which cause the model to function the same way. For example, a traffic intersection model might specify that the probability of a left turn in a three-way turning lane is 0.2, the probability of a right turn is 0.35, and the probability of continuing straight is 0.45. The modeler incorporates this into the model using a series of conditional branches which branch on a value produced by a random number generator. The random number generator produces numbers in the range $0 \leq rn < 1$ (ignoring hardware limitations). In effect, the model contains three separate equivalence partitions here: $0 \leq rn < 0.2$, $0.2 \leq rn < 0.55$, and $0.55 \leq rn < 1$. Each test case from within a given partition (i.e., class) will have the same effect on the model.

Boundary analysis is a technique that tests the activity of the model using test cases on the boundaries of input equivalence partitions. Test cases are generated just within, on top of, and just outside of the partition boundaries [Myers 1979]. In the example given above, rather than arbitrarily selecting test cases from each of these equivalence classes, the modeler would, using boundary analysis, generate test cases at the edges of each class. Such test cases might be 0.0, ± 0.000001, 0.199999, 0.2, and so on. In addition to generating test data on the basis of input equivalence classes, it is also useful to generate test data which will cause the model to produce values on the boundaries of output equivalence classes [Myers 1979]. The underlying rationale for this technique as a whole is that the most error-prone test cases lie along the boundaries [Ould and Unwin 1986]. This meets the objective of identifying "interesting" test cases mentioned in Chapter 1. Notice that an invalid value was among the test values listed in the example above. This relates directly to the concept discussed in stress testing (Section 4.3.5).

The primary difficulty in boundary analysis lies in determining the boundaries of the equivalence classes. The example above was trivial. Typical, "real-life" simulation models will involve much more effort to establish their boundaries, with each model having its own special conditions to consider.

4.5.4 Advantages and Disadvantages of Constraint Analysis

Constraint analysis techniques find their origins in the predicate calculus. The assertions themselves are model predicates. The activity between entry and exit assertions is the transformation of the predicates. However, the ability to state and place assertions effectively relies in large part on formal model specification. Creating a formal specification is a difficult task. Using assertions is further complicated by the lack of assertion capabilities in programming languages. Most languages provide no facility for performing assertion checking. Yet another drawback is high human resource cost. Likewise, computer resource cost is very high, primarily because the instrumented model suffers performance degradation.

On the other hand, constraint analysis is a very effective method of verification. Assertions placed in the source code provide a good source of documentation. Further, constraint analysis can actually verify that the model (or some subset thereof) is functioning correctly, i.e., in accordance with its specification. This is *essential* in simulation studies.

The programmed model is not simply a software package designed to provide some range of capability. It is a representation of a real entity. It is designed to provide information about the system it represents. It must not simply function correctly, it must produce sufficiently valid results, as stated in its specification. Consider, for example, a decision-support system used by a stock market analyst to make predictions and recommendations concerning market activity. When the analyst needs help in interpreting data, he consults the system, which re-

sponds with the appropriate interpretation *on the basis of an underlying model.* Suppose, however, that some constraint of the model is violated. The decision-support program may function correctly by evaluating the data in the proper manner. Unfortunately, the results given are invalid—mathematically correct, but invalid for the given model. From this it is seen that the programmed model has an added dimension of verification need.

This leads to another aspect of the programmed model not common to other software. The execution lifetime of the programmed model is spent in the Experimentation process. Its on-going purpose is to represent some other entity for the sake of making statements about that entity. A single violated assumption, undetected, invalidates the entire study. Obviously, this is not the same situation as with the employee who gets a garbage paycheck because of a bad input. The employee will quickly point out that some assumption was violated. The simulation model will not. It is clear that PMV needs are high. Using the assertion checking technique, the modeler can be assured that the model is operating within its bounds; conversely, when it is not, the modeler will know. By being able to make this statement, the modeler builds evidence to support acceptance of the model.

The ramifications of not verifying adherance to assumptions are too great for the serious modeler not to employ constraint analysis. The claim that the technique is too expensive is simply not justified. Computer resource cost has long since exceeded human resource cost. Constraint analysis may require more execution time, but it more than makes up for it: (1) by reducing the need to iterate back through the life cycle (e.g., to redo an entire set of experiments) and (2) by enhancing the credibility of the model.

4.6 Formal Analysis

Formal analysis completes the spectrum of PMV methods. Formal analysis is, as the name implies, based on formal mathematical proof of correctness. If attainable, formal proof of correctness is the most effective means of verifying software. Unfortunately, "if attainable" is the overriding point with regard to formal analysis. Current state-of-the-art model proving techniques are simply not capable of being applied to even the simplest general modeling problems. However, formal techniques serve as the foundation for other verification techniques and will be covered here for the sake of completeness. Among the prevalent terms heard when mentioning formal analysis are proof of correctness, predicate calculus, predicate transformation, λ-calculus, inference, logical deduction, and induction.

The use of the term *correct* with respect to PMV and software verification in general is a relative rather than absolute term. When one speaks of *model correctness*, he means that the model meets its specifications. Formal *proof of correctness* corresponds to expressing the model in a precise notation and then mathematically proving (1) that the executed model terminates and (2) that it satisfies the requirements of its specification [Backhouse 1986].

The *λ-calculus* [Church 1951; Stoy 1977; Barendregt 1981] is a system for transforming the programmed model into formal expressions. The λ-calculus is a string-rewriting system and the model itself can be considered as a large string. The λ-calculus specifies rules for rewriting strings, i.e., the model, into λ-calculus expressions. Using the λ-calculus, the modeler can formally express the model so that mathematical proof techniques can be applied.

The *predicate calculus* provides rules for manipulating predicates. A predicate is a combination of simple relations, such as *completed_jobs* > *steady_state_length*. A predicate will either be true or false. The programmed model can be defined in terms of predicates and manipu-

lated using the rules of the predicate calculus. The predicate calculus forms the basis of all formal specification languages [Backhouse 1986]. *Predicate transformation* [Dijkstra 1975] provides a basis for verifying model correctness by formally defining the semantics of the model with a mapping which transforms model output states to all possible model input states. This representation provides the basis for proving whether or not the model is correct (if it has transformed initial states to termination states properly).

Inference, logical deduction, and *induction* are simply acts of justifying conclusions on the basis of premises given. An argument is valid if the steps used to progress from the premises to the conclusion conform to established *rules of inference.* (These rules were developed by the German mathematician Gentzen). Inductive reasoning is based on invariant properties of a set of observations (assertions are invariants since their value is defined to be true). A typical inductive argument would be one similar to the one given in the previous section for inductive assertions: given that the initial model assertion is correct, it stands to reason that if each path progressing from that assertion can be shown to be correct, and subsequently each path progressing from the previous assertion is correct, etc., then the model must be correct if it terminates. There are formal induction proof techniques for the intuitive explanation just given.

Several authors provide detailed coverage of formal analysis, among which are [Berg et al. 1982; Backhouse 1986; Dijkstra 1976; Hoare 1969; Knuth 1968,1969; Polya 1954; Stoy 1977; Yeh et al. 1977].

4.6.1 Advantages and Disadvantages of Formal Analysis

Attaining proof of correctness in a realistic sense is not possible with current technology. The complexity of the task is simply too great. Setting up a proof for even a simple model is an

expensive, time-consuming undertaking. Completing the proof would be just as intense. The matter is further complicated by non-mathematical considerations such as machine dependencies and other related idiosynchrosies. However, the advantage of realizing proof of correctness—*complete* PMV—is so great that when the capability is realized, it will revolutionize the verification of software.

5.0 CONCLUDING REMARKS AND FUTURE RESEARCH

It has been shown that there is a definite problem within the simulation community concerning PMV. There is a lack of understanding and appreciation of PMV, and there is a shortage in the literature (as discussed in Chapters 1 and 3) of techniques and guidelines for performing PMV.

For many modelers, the verification process ends the moment the model specification is relegated to the software engineering group for programming, then validation resumes when the programmed model is returned for experimentation. Unfortunately, the modeler and the programming team each have their own (colliding) assumptions about how the verification is to be managed. The lack of communication between the two groups is one of the major contributors to increased testing requirements and cost. For modelers who must create the programmed model themselves, PMV is simply viewed as debugging the code. This view has been shown to be extremely inaccurate.

While there is ample literature on software verification, that literature is targeted towards the software engineer. The overwhelming majority of simulation practitioners are ***not*** software engineers, do not speak the software engineering "language," and thus do not reap the benefits of verification technology available from the software engineering field. Not only is the current simulation community affected, newcomers to the field are affected by not getting ad-

equate exposure to PMV. The modeler needs to recognize the full scope of the PMV process, needs techniques which satisfy PMV needs, and needs guidelines for applying the techniques to perform PMV.

This thesis fills those voids in a number of ways, the first by contributing the Taxonomy for Programmed Model Verification Methods. The taxonomy provides a comprehensive picture of the PMV process; the modeler can quickly grasp the scope of PMV. The taxonomy is more than just a two-dimensional picture of verification techniques. It is actually a six-dimensional depiction of the verification domain. This multi-dimensionality occurs because of the potential overlap of a technique into several categories. For example, it is possible for a technique to be informal, static, and symbolic all at the same time, such as the intuitive reduction of model structure using its symbolic execution tree. The expected effectiveness and power of a technique will generally increase as the technique falls within more formal categories of the taxonomy. Formality—and corresponding verification effectiveness—increases from left to right across the taxonomy; likewise, one would expect the cost to increase from left to right, which it does. Using the taxonomy, the modeler can identify techniques with which to perform PMV, and is illuminated to the character of, the relationships among, and the advantages and disadvantages of the verification techniques. This is helpful not only in applying individual techniques, but also in providing guidance for planning and directing the PMV process. The taxonomy provides a very broad base from which the modeler can establish expansive evidence of model credibility. Such a resource as this taxonomy has not heretofore been provided in the literature.

The extensive review of software verification techniques—adapted to the terminology of the simulation modeler—provides additional substance to the taxonomy by explaining in familiar terms the mechanics of the various verification techniques. The simulation case study provides a concrete example of the use of the various techniques to perform PMV. The case study provides a pragmatic illustration of PMV in operation. Even the many software devel-

opers who have yet to understand the software verification process will find these resources invaluable.

The emphasis of this thesis is on performing PMV in an effort to increase model credibility. This thesis fills a real void that exists in the simulation model life cycle literature. It is important to note, however, that PMV is but one of many CASs in the simulation model life cycle. It cannot be emphasized enough that the other aspects of the life cycle must not be overlooked. Further, as has been alluded to several times throughout this thesis, the earlier in the life cycle that errors can be detected, the less verification cost and effort will be required in subsequent stages of the life cycle. Nance and Overstreet [1987], Overstreet and Nance [1985], Balci and Nance [1987], Balci [1986], and Overstreet [1982] deal with this topic, as do others. In the utopian sense, the automation-based paradigm [Balzer et al. 1983] would vastly reduce (if not eliminate) the need for PMV by providing the ability to generate the model directly from the model specification. Until that paradigm is realized, however, PMV is an inevitable step of the life cycle of a simulation study, and this thesis provides guidelines needed to make that process understood and manageable.

The taxonomy developed in this paper provides a bridge for the simulation community to share the wealth of verification technology available in the software engineering domain. The taxonomy and techniques presented herein provide a clear view of how to approach programmed model verification in terms that the modeler is comfortable with. Additionally, the modeler is given insight into how to apply the verification techniques for assessing the credibility of simulation models. Future research will be directed towards the task of identifying techniques which are uniquely applicable to PMV, and determining unique properties of, or special means of applying, techniques for the verification of simulation models.

BIBLIOGRAPHY

Ackerman, A.F., P.J. Fowler, and R.G. Ebenau (1983), "Software Inspections and the Industrial Production of Software," In ***Software Validation: Inspection, Testing, Verification, Alternatives,*** Proceedings of the Symposium on Software Validation (Darmstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen (Ed.), pp. 13-40.

Adrion, W.R., M.A. Branstad and J.C. Cherniavsky (1982), "Validation, Verification, and Testing of Computer Software," ***Computing Surveys 14,*** 2 (June), 159-192.

Allen, F.E. and J. Cocke (1976), "A Program Data Flow Analysis Procedure," ***Communications of the ACM 19,*** 3 (Mar.), 137-147.

Andrews, D.M. and J.P. Benson (1979), "Using Executable Assertions for Testing," ***Proceedings of the Thirteenth Asilomar Conference on Circuits, Systems, and Computers,*** Pacific Grove, Calif., pp. 302-305.

Arthur, J.D. and R.E. Nance (1987), "Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products," Technical Report TR 87-16, Department of Computer Science, Virginia Tech, Blacksburg, Va., Apr.

Arthur, J.D., R.E. Nance, and S.M. Henry (1986), "A Procedural Approach to Evaluating Software Development Methodologies: The Foundation," Technical Report TR 86-24, Department of Computer Science, Virginia Tech, Blacksburg, Va., Sept.

Backhouse, R.C. (1986), ***Program Construction and Verification,*** Prentice-Hall International (UK) Ltd, Great Britain.

Balci, O. (1986), "Requirements for Model Development Environments," ***Computers and Operations Research 13***, 1 (Jan.-Feb.), 53-67.

Balci, O. (1987), "Guidelines for Successful Simulation Studies: Part I and II," Technical Report TR-85-2, Department of Computer Science, Virginia Tech, Blacksburg, Va., Mar.

Balci, O. (1988), "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages," ***Proceedings of the 1988 Winter Simulation Conference*** (San Diego, Calif., Dec. 12-14), to appear.

Balci, O. and R.E. Nance (1985), "Formulated problem verification as an explicit requirement of model credibility," ***Simulation 45,*** 2 (Aug.), 76-86.

Balci, O. and R.E. Nance (1987), "Simulation Support: Prototyping the Automation-Based Paradigm," Technical Report TR 87-20, Department of Computer Science, Virginia Tech, Blacksburg, Va., July.

Balci, O. and R.G. Sargent (1981), "A Methodology for Cost-Risk Analysis in the Statistical Validation of Simulation Models," ***Communications of the ACM 24***, 4 (Apr.), 190-197.

Balci, O. and R.G. Sargent (1984), "A Bibliography on the Credibility Assessment and Validation of Simulation and Mathematical Models," ***Simuletter 15***, (July), 15-27.

Balzer, R.M. (1969), "EXDAMS - Extended Debugging and Monitoring System," ***Proceedings of the AFIPS Sprint Joint Computer Conference,*** Vol. 34, pp. 567-580.

Balzer, R.M., T.E. Cheatham, and C. Green (1983), "Software Technology in the 1990's: Using a New Paradigm," ***Computer 16***, 11 (Nov.), 39-45.

Barendregt, H.P. (1981), ***The Lambda Calculus: Its Syntax and Semantics***, North-Holland Publishing Company, New York, N.Y.

Basu, S.K. and R.T. Yeh (1975), "Strong Verifications of Programs," ***IEEE Transactions of Software Engineering SE-1***, 3, 339-346.

Beizer, B. (1983), ***Software Testing Techniques*** Van Nostrand-Reinhold, New York, N.Y.

Beizer, B. (1984), ***Software System Testing Techniques and Quality Assurance,*** Van Nostrand-Reinhold, New York, N.Y.

Berg, H.K, W.E. Boebert, W.R. Franta, and T.G. Moher (1982), ***Formal Methods of Program Verification and Specification,*** Prentice-Hall, Englewood Cliffs, N.J.

Boehm, B.W. (1976), "Software Engineering," ***IEEE Transactions on Computers C-25***, 12 (Dec.), 1266-1241.

Boehm, B.W. (1984), "Verifying and Validating Software Requirements and Design Specifications," ***IEEE Software 1,*** 1 (Jan.), 75-88.

Boehm, B.W. (1986), "A Spiral Model of Software Development and Enhancement," ***ACM Software Engineering Notes 11,*** 4 (Aug.), 14-24.

Boehm, B.W. (1988), "A Spiral Model of Software Development and Enhancement," ***IEEE Computer 21,*** 5 (May), 61-72.

Bohm, C. and G. Jacopini (1966), "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," ***Communications of the ACM 9,*** 5 (May), 366-371.

Boyer, R.S., B. Elspas, and K.N. Levitt, "SELECT: A Formal System for Testing and Debugging Programs by Symbolic Execution," ***Proceedings of the International Conference on Reliable Software, SigPlan Notices 10,*** 6, 234-245.

Branstad, M.A., J.C. Cherniavsky, and W.R. Adrion (1980), "Validation, Verification, and Testing for the Individual Programmer," In ***Structured Testing,*** T.J. McCabe, Ed., IEEE Computer Society Press, Silver Sprint, Md., pp. 57-78.

Brooks, F. (1975), ***The Mythical Man-Month,*** Addison-Wesley, Reading, Mass.

Bryan, W.E. and S.G. Siegel (1987), "Software Configuration Management—A Practical Look," In ***Handbook of Software Quality Assurance,*** G.G. Schulmeyer and J.I. McManus, Eds., Van Nostrand-Reinhold Company, New York, N.Y., pp. 211-247.

Buckles, B.P. and J.P. Ryan (1977), "Higher Level Verification," ***Proceedings of the Summer Computer Simulation Conference,*** Chicago, Ill., pp. 735-739.

Buck, R.D. and J.H. Dobbins (1983), "Application of Software Inspection Methodology in Design and Code," In ***Software Validation: Inspection, Testing, Verification, Alternatives,*** Proceedings of the Symposium on Software Validation (Darmstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen (Ed.), pp. 41-56.

Budd, T.A., R.A. DeMillo, R.J. Lipton, and F.G. Sayward (1980), Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," ***Seventh Annual ACM Symposium on Principles of Programming Languages,*** (Jan.), pp. 220-223.

Chandrasekharan, M., B. Dasarathy, and Z. Kishimoto (1985), "Requirements-Based Testing of Real-Time Systems: Modeling for Testability," ***IEEE Computer 18,*** 4 (Apr.), 71-80.

Charlton, C.C. and P.H. Leng (1983), "Aids for Pragmatic Error Detection," ***Software - Practice and Experience 13,*** 59-66.

Cherniavsky, J.C. and C.H. Smith (1987), "A Recursion Theoretic Approach to Program Testing," ***IEEE Transactions on Software Engineering SE-13,*** 7 (July), 777-784.

Chow, R.S. (1976), "A Generalized Assertion Language," ***Proceedings of the Second International Conference on Software Engineering,*** San Francisco, Calif., pp. 392-399.

Church, A. (1951), "The Calculi of Lambda-Conversion," ***Annals of Mathematical Studies, Vol. 6,*** Princeton University Press, Princeton, N.J.

Chusho, T. (1987), "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing," ***IEEE Transactions on Software Engineering SE-13,*** 5 (May), 509-517.

Chu, Y. (1976), "Introducing a Software Design Language," ***Proceedings of the 2nd International Conference on Software Engineering,*** 297-304.

Clarke, L.A. (1976), "A System to Generate Test Data and Symbolically Execute Programs," ***IEEE Transactions on Software Engineering SE-2,*** 3 (Sept.), 215-222.

Clarke, L.A. and D.J. Richardson (1983), "Symbolic Execution — An Aid to Testing and Verification," In ***Software Validation: Inspection, Testing, Verification, Alternatives,*** Proceedings of the Symposium on Software Validation (Darmstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen (Ed.), pp. 141-166.

Clarke, S. (1988), "The Increasing Importance of Formal Methods," ***Computer Bulletin 4,*** 1 (Mar.), 22-26.

Crocker, S.D. (1985), "Engineering Requirements for Production Quality Verification Systems," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 15-16.

Darringer, J.A. and J.C. King (1978), "Applications of Symbolic Execution to Program Testing," ***Computer 11,*** 4 (Apr.), 309-317.

Davis, M. and E. Weyuker (1988), "Metric Space-based Test-data Adequacy Criteria," ***The Computer Journal 31,*** 1, 17-24.

Dershowity, N. (1985), "Rewriting and Verification," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 60.

Deutsch, M.S. (1981), "Software Project Verification and Validation," ***Computer 14,*** 4, 54-70.

Deutsch, M.S. (1982), ***Software Verification and Validation: Realistic Project Approaches,*** Prentice-Hall, Englewood Cliffs, N.J.

DeMarco, T. (1978), ***Structured Analysis and System Specification,*** Yourdon, Inc., New York, N.Y.

DeMarco, T. (1979), ***Concise Notes on Software Engineering,*** Yourdon Press, New York, N.Y.

DeMillo, R.A., R.J. Lipton, and F.G. Sayward (1978), "Hints on Test Data Selection: Help for the Practicing Programmer," ***Computer 11,*** 4 (Apr.), 34-41.

Dieudonne, J.E. (1979), "Verification and Validation of the NASA Terminal Configured Vehicle's (TCV) Wind Analysis Program Using Real-Time Digital Simulation," ***Proceedings of the Summer Simulation Conference,*** Toronto, Canada, pp. 584-590.

Dijkstra, E.W. (1975), "Guarded Commands, Non-determinacy and a Calculus for the Derivation of Programs," ***Communications of the ACM 18***, 8 (Aug.), 453-457.

Dijkstra, E.W. (1976), ***A Discipline of Programming,*** Prentice-Hall, Englewood Cliffs, N.J.

Dobbins, J.H. (1987), "Inspections as an Up-Front Quality Technique," In ***Handbook of Software Quality Assurance,*** G.G. Schulmeyer and J.I. McManus, Eds., Van Nostrand-Reinhold Company, New York, N.Y., pp. 137-177.

Dobbins, J.H. and R.D. Buck (1987), "The Cost of Software Quality," In ***Handbook of Software Quality Assurance,*** G.G. Schulmeyer and J.I. McManus, Eds., Van Nostrand-Reinhold Company, New York, N.Y., pp. 119-136.

Doe, D.D. and E.H. Bersoff (1986), "The Software Productivity Consortium (SPC): An Industry Initiative to Improve the Productivity and Quality of Mission-Critical Software," ***The Journal of Systems and Software,*** 6, 367-378.

Dunn, R.H. (1987), "The Quest for Software Reliability," In ***Handbook of Software Quality Assurance,*** G.G. Schulmeyer and J.I. McManus, Eds., Van Nostrand-Reinhold Company, New York, N.Y., pp. 342-384.

Elmendorf, W.R. (1973), "Cause-effect Graphs in Functional Testing," Technical Report TR-00.2487, IBM Systems Development Division, Poughkeepsie, N.Y.

Elspas, B., K.N. Levitt, R.J. Waldinger, and A. Waksman (1972), "An Assessment of Techniques for Proving Program Correctness, " ***Computing Surveys 4,*** 2.

Enos, J.C. and R.R. Willis (1978), "Tools for Embedded Software Design Verification," In ***Tools for Embedded Computing Systems Software,*** NASA Conference Publication 2064, pp. 93-96.

Fagan, M.E. (1976), "Design and Code Inspections to Reduce Errors in Program Development," ***IBM Systems Journal 15,*** 3, 1976, 182-211.

Fagan, M.E. (1977), "Inspecting Software Design and Code," ***Datamation,*** (Oct.), 133-144.

Fagan, M.E. (1986), "Advances in Software Inspections," ***IEEE Transactions on Software Engineering SE-12,*** 7 (July), 744-751.

Fairley, R.E. (1975), "An Experimental Program-Testing Facility," ***IEEE Transactions on Software Engineering SE-1,*** 4, 350-357.

Fairley, R.E. (1976), "Dynamic Testing of Simulation Software," In ***Proceedings of the 1976 Summer Computer Simulation Conference*** (Washington, D.C., July 12-14). Simulation Councils, La Jolla, Calif., pp. 708-710.

Fairley, R.E. (1977), "A New Approach to Software Verification and Validation," ***Proceedings of the 1977 Summer Computer Simulation Conference,*** Chicago, Ill., pp. 709-712.

Fairley, R.E. (1978), "Tutorial: Static Analysis and Dynamic Testing of Computer Software," ***Computer 11,*** 4 (Apr.), 14-23.

Fishman, G.S. (1973), ***Concepts and Methods in Discrete Event Digital Simulation***, Wiley-Interscience, New York, N.Y.

Floyd, R.W. (1967), "Assigning Meaning to Programs," ***Proceedings of the American Mathematical Society Symposium in Applied Mathematics***, 19, pp. 19-31. Fujii, M.S. and M.A. Ikezawa (1978), "Use of Symbolic Execution in Verification and Validation," In ***Tools for Embedded Computing Systems Software,*** NASA Conference Publication 2064, pp. 113-116.

Gabow, H.N., S.N. Maheshwari, and L.J. Osterweil (1976), "On Two Problems in the Generation of Program Test Paths," ***IEEE Transactions on Software Engineering SE-2,*** 3 (Sept.), 227-231.

Garcia-Molina, H., F. Germano, and W.H. Kohler (1984), "Debugging a Distributed Computing System," ***IEEE Transactions on Software Engineering SE-10,*** 2, 210-219.

General Accounting Office (1987), "DOD SIMULATIONS: Improved Assessment Procedure Would Increase the Credibility of Results", GAO/PEMD-88-3, U.S. G.A.O., Washington, D.C., Dec.

Golub, J. and D. Hill (1987), "TIMES (Terminal Interactive Menu Execution System) as a Simulation Validation Tool," In ***Proceedings of the 1987 Summer Computer Simulation Conference*** (Montreal, Quebec, Canada, July 27-30). Simulation Councils, La Jolla, Calif., pp. 123-127.

Goodenough, J.B. and S.L. Gerhart (1977), "Toward a Theory of Testing: Data Selection Criteria," In ***Current Trends in Programming Methodology, Vol. 2,*** R. Yeh (editor), Prentice-Hall, Englewood Cliffs, N.J., pp. 44-79.

Good, D.I., R.S. Boyer, and J.S. Moore (1985), "A Second Generation Verification Environment," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 48.

Grady, R.B. and D.L. Caswell (1987), ***Software Metrics: Establishing a Company-Wide Program***, Prentice-Hall, Englewood Cliffs, N.J.

Gunther, R.C. (1978), ***Management Methodology for Software Product Engineering,*** John Wiley & Sons, New York, N.Y.

Halstead, M.H. (1977), ***Elements of Software Science,*** Elsevier North-Holland, Inc., New York, N.Y.

Haung, J.C. (1975), "An Approach to Program Testing," ***Computing Surveys 7,*** 3, 318-333.

Haung, J.C. (1977), "Error Detection Through Program Testing," In ***Current Trends in Programming Methodology, Vol. 2,*** R. Yeh (editor), Prentice-Hall, Englewood Cliffs, N.J., pp. 16-43.

Haung, J.C. (1978), "Program Instrumentation and Software Testing," ***Computer 11,*** 4 (Apr.), 25-32.

Haung, J.C. (1980), "Instrumenting Programs for Symbolic-Trace Generation," ***IEEE Computer 13,*** 12 (Dec.), 17-23.

Hausen, H. (1983), "Comments on Practical Constraints of Software Validation Techniques," In ***Software Validation: Inspection, Testing, Verification, Alternatives,*** Proceedings of the Symposium on Software Validation (Darmstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen (Ed.), pp. 323-333.

Hausen, H.L. and M. Mullerburg (1983), "An Introduction to Quality Assurance and Control of Software," In ***Software Validation: Inspection, Testing, Verification, Alternatives,*** Proceedings of the Symposium on Software Validation (Darmstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen (Ed.), pp. 3-9.

Henderson, P. (1977), "Structure Program Testing," In ***Current Trends in Programming Methodology, Vol. 2,*** R. Yeh (editor), Prentice-Hall, Englewood Cliffs, N.J., pp. 1-15.

Hetzel, W.C. (1973), ***Program Test Methods***, Prentice-Hall, Englewood Cliffs, N.J.

Hoare, C.A.R. (1969), "An Axiomatic Basis for Computer Programming," ***Communications of the ACM 12,*** 10 (Oct.), 576-583.

Hoffman, K.J., T.H. Andres, and J.A.K. Reid (1984), "Experience in the Application of Quality Assurance Techniques to a Scientific Simulation Code," ***Proceedings of the 1984 Summer Computer Simulation Conference*** (Boston, Mass., July 23-25), Simulation Councils, La Jolla, Calif., pp. 246-251.

Hoffnagle, G.F. and W.E. Beregi (1985), "Automating the software development process," ***IBM Systems Journal 24,*** 2, 102-120.

Hollocker, C.P. (1987), "The Standardization of Software Reviews and Audits," In ***Handbook of Software Quality Assurance,*** G.G. Schulmeyer and J.I. McManus, Eds., Van Nostrand-Reinhold Company, New York, N.Y., pp. 211-266.

Holmes, W.M. (1984), "Validation and Credible Models for System Simulation," In ***Proceedings of the 1984 Summer Computer Simulation Conference*** (Boston, Mass., July 23-25). Simulation Councils, La Jolla, Calif., pp. 239-245.

Hookway, R.J. (1985), "Verifying Ada Programs," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 51-52.

Hopcroft, J.E. and J.O. Ullman (1969), ***Formal Languages and Their Relations to Automata,*** Addison-Wesley, Reading, Mass.

Horowitz, E. and R.C. Williamson (1986), "SODOS: A Software Documentation Support Environment—Its Definition," ***IEEE Transactions on Software Engineering SE-12,*** 8 (Aug.), 849-859.

Howden, W.E. (1976), "Reliability of the Path Analysis Testing Strategy," ***IEEE Transactions on Software Engineering SE-2,*** 3 (Sept.), 208-214.

Howden, W.E. (1977), "Symbolic Testing and the DISSECT Symbolic Evaluation System," ***IEEE Transactions on Software Engineering SE-3,*** 4 (July), 266-278.

Howden, W.E. (1980), "Functional Program Testing," ***IEEE Transactions on Software Engineering SE-6,*** 2, 162-169.

Howden, W.E. (1982), "Validation of Scientific Programs," ***Computing Surveys 14,*** 2 (June), 193-227.

Howden, W.E. (1985), "The Theory and Practice of Functional Testing," ***IEEE Software 2,*** 5, 6-17.

Howden, W.E. (1986), "A Functional Approach to Program Testing and Analysis," ***IEEE Transactions on Software Engineering SE-12,*** 10 (Oct.), 997-1005.

Innis, G.S., S. Schlesinger, and R.J. Sylvester (1977), "Model Certification — Varying Views from Different Specialties," In ***Proceedings of the 1977 Summer Computer Simulation Conference,*** Chicago, Ill., pp. 695-698.

IEEE (1983), "IEEE Standard Glossary of Software Engineering Terminology," ***Publication 729-1983,*** The Institute of Electrical and Electronics Engineers, New York, N.Y.

Keller, R.M. (1976), "Formal Verification of Parallel Programs," ***Communications of the ACM 19,*** 7 (July), 371-384.

King, J.C. (1976), "Symbolic Execution and Program Testing," ***Communications of the ACM 19,*** 7 (July), 385-394.

Kirchoff, M.K. and S.H. Dalrymple (1978), "Instrumentation and Control of a Virtual Machine," In ***Tools for Embedded Computing Systems Software,*** NASA Conference Publication 2064, pp. 11-14.

Knuth, D.E. (1968), ***The Art of Computer Programming Vol. I: Fundamental Algorithms,*** Addison-Wesley, Reading, Mass.

Knuth, D.E. (1969), ***The Art of Computer Programming Vol. II: Seminumerical Algorithms,*** Addison-Wesley, Reading, Mass.

Koskossidis, D.A. and B.J. Davies (1984), "Validation and Verification of Job Shop Simulation Models," In ***Proceedings of the 1984 Summer Simulation Conference*** (Boston, Mass., July 23-25). Simulation Councils, La Jolla, Calif., pp. 252-255.

Landwehr, C.E. (1985), "Does Program Verification Help? How Much?," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 107.

Lauesen, S. (1979), "Debugging Techniques," ***Software - Practice and Experience 9,*** 51-63.

Leavenworth, B.M. (1977), "Structured Debugging Using a Domain Specific Language," ***Software - Practice and Experience 7,*** 475-482.

Lipton, R.J., (1975), "Reduction: A Method of Proving Properties of Parallel Programs," ***Communications of the ACM 18,*** 12 (Dec.), 717-721.

London, R.L. (1977), "Perspectives on Program Verification," In ***Current Trends in Programming Methodology, Vol. 2,*** R. Yeh (editor), Prentice-Hall, Englewood Cliffs, N.J., pp. 151-172.

Macdonald, R. (1985), "Verifying a Real System Design - Some of the Problems," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 126-129.

Manna, Z., S. Ness, and J. Vuillemin (1973), "Inductive Methods for Proving Properties of Programs," ***Communications of the ACM 16***, 8 (Aug.), 491-502.

Marcus, L., S.D. Crocker, and J.R. Landauer (1985), "SDVS: A System for Verifying Microcode Correctness," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 7-14.

Marks, H. (1979), "PROBE/1, A Tool for Verifying Software," ***Proceedings of the 1979 Summer Computer Computer Simulation Conference,*** Toronto, Ontario, Canada, pp. 635-638.

Matthews, R.S., K.H. Muralidhar, and S. Sparks (1988), "MAP 2.1 Conformance Testing Tools," ***IEEE Transactions on Software Engineering 14,*** 3 (Mar.), 363-374.

Miller, E. and W.E. Howden (editors) (1978), ***Tutorial: Software Testing and Validation Techniques,*** IEEE Computer Society, Catalog EHO 138-8, Long Beach, Calif.

Miller, E.F. (1977), "Modern Program Verification Techniques," ***Proceedings of the Summer Computer Simulation Conference,*** Chicago, Ill., pp. 721-726.

Miller, E.F. and R.A. Melton (1975), "Automated Generation of Testcase Datasets," ***Proceedings of the International Conference on Reliable Software, SigPlan Notices 10,*** 6, 113-128.

Mills, H.D. (1988), "Stepwise Refinement and Verification in Box-Structured Systems, " ***IEEE Computer 21,*** 6 (June), 23-36.

Moose, R.L. and R.E. Nance (1985), "Model Analysis in a Model Development Environment," Technical Report TR-85-27, Department of Computer Science, Virginia Tech, Blacksburg, Va.

Morris, J.H. Jr. and B. Wegbreit (1977), "Program Verification by Subgoal Induction," In ***Current Trends in Programming Methodology, Vol. 2,*** R. Yeh (Ed.), Prentice-Hall, Englewood Cliffs, N.J., pp. 197-227.

Myers, G.J. (1975), ***Reliable Software Through Composite Design,*** Mason/Charter Publishers, Inc., New York, N.Y.

Myers, G.J. (1978), "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," ***Communications of the ACM 21,*** 9 (Sept.), 760-768.

Myers, G.J. (1978a), ***Composite/Structured Design,*** Van Nostrand-Reinhold Company, New York, N.Y.

Myers, G.J. (1979), ***The Art of Software Testing,*** John Wiley & Sons, New York, N.Y.

Nance, R.E. (1981), "Model Representation in Discrete Event Simulation: The Conical Methodology," Technical Report CS81003-R, Department of Computer Science, Virginia Tech, Blacksburg, Va., Mar.

Nance, R.E. and C.M. Overstreet (1987), "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," ***Transactions of the Society for Computer Simulation 4,*** 1 (Jan.), 33-57.

Nelson, E. (1978), "Software Testing vs. Formal Verification," In ***Proceedings of the 1978 Summer Computer Simulation Conference,*** Newport Beach, Calif., pp. 744-747.

O'Keefe, R.M., O. Balci, and E.P. Smith (1987), "Validating Expert System Performance," ***IEEE Expert,*** Winter, 81-90.

Osterweil, L. (1983), "Integrating the Testing, Analysis and Debugging of Programs," In ***Software Validation: Inspection, Testing, Verification, Alternatives,*** Proceedings of the Symposium on Software Validation (Darmstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen (Ed.), pp. 73-101.

Osterweil, L.J. and L.D. Fosdick (1976), "DAVE — A Validation Error Detection and Documentation System for FORTRAN Programs," ***Software Practice and Experience***, 6, 473-486.

Ould, M.A. and C. Unwin (1986), ***Testing in Software Development,*** Cambridge University Press, Great Britain.

Overstreet, C.M. (1982), "Model Specification and Analysis for Discrete Event Simulation," Ph.D. Dissertation, Virginia Tech, Blacksburg, Va.

Overstreet, C.M. and R.E. Nance (1985), "A Specification Language to Assist Analysis of Discrete Event Simulation Models," ***Communications of the ACM 28***, 2 (Fed.), 190-201.

Overstreet, C.M., R.E. Nance, O. Balci, and L.F. Barger (1986), "Specification Languages: Understanding Their Role in Simulation Model Development," Technical Report TR 87-7, Department of Computer Science, Virginia Tech, Blacksburg, Va., Dec.

Owicki, S. and D. Gries (1976), "Verifying Properties of Parallel Programs: An Axiomatic Approach," ***Communications of the ACM 19,*** 5 (May), 279-285.

Pace, D.K. (1987), "Model and Simulation Design-and-Development Procedures to Enhance Validation and Credibility," In ***Proceedings of the 1987 Summer Computer Simulation Conference*** (Montreal, Quebec, Canada, July 27-30). Simulation Councils, La Jolla, Calif., pp. 138-141.

Panzl, D.J. (1976), "Test Procedures: A New Approach to Software Verification," In ***Proceedings of the 2nd International Conference on Software Engineering,*** San Francisco, Calif., pp. 477-485.

Panzl, D.J. (1978), "Automatic Software Test Drivers," ***Computer 11,*** 4 (Apr.), 44-50.

Parnas, D.L. (1979), "Designing Software for Ease of Extension and Contraction," ***IEEE Transactions on Software Engineering SE-5,*** 2 (Mar.), 128-137.

Ploedereder, E. (1983), "Symbolic Evaluation as a Basis for Integrated Validation," In ***Software Validation: Inspection, Testing, Verification, Alternatives,*** Proceedings of the Symposium on Software Validation (Darmstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen (Ed.), pp. 167-185.

Polya, G. (1954), ***Induction and Analogy in Mathematics,*** Princeton University Press, Princeton, N.J.

Poston, R. (1988), "Preventing most-probably errors in testing," ***IEEE Software 5,*** 2 (Mar.), 86-88.

Powell, P.B., Ed. (1982), "Planning for Software Validation, Verification, and Testing," ***National Bureau of Standards Special Publication 500-98,*** Washington, D.C.

Prather, R.E. and J.P. Myers, Jr. (1987), "The Path Prefix Software Testing Strategy," ***IEEE Transactions on Software Engineering SE-13,*** 7 (July), 761-766.

Pressman, R.S. (1987), ***Software Engineering: A Practitioner's Approach,*** 2nd Ed., McGraw-Hill, New York, N.Y.

Probert, R.L. (1982), "Optimal Insertion of Software Probes in Well-Delimited Programs," ***IEEE Transactions on Software Engineering SE-8,*** 1, 34-42.

Ramamoorthy, C.V. and S.F. Ho (1977), "Testing Large Software with Automated Software Evaluation Systems," In ***Current Trends In Programming Methodology, Vol. 2,*** R. Yeh (Ed.), Prentice-Hall, Englewood Cliffs, N.J., pp. 112-150.

Ramamoorthy, C.V., S.F. Ho, and W.T. Chen (1976), "On the Automated Generation of Program Test Data," ***IEEE Transactions on Software Engineering SE-2,*** 4 (Dec.), 293-300.

Razaz, M. and J.L. Schonfelder (1988), "Test Procedures for Measurement of Floating-Point Characteristics of Computing Environments," ***The Computer Journal 31,*** 1, 12-16.

Remus, H. (1983), "Integrated Software Validation in the View of Inspections/Reviews," In ***Software Validation: Inspection, Testing, Verification, Alternatives,*** Proceedings of the Symposium on Software Validation (Darmstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen (Ed.), pp. 57-64.

Reynolds, C. and R.T. Yeh (1976), "Induction as the Basis for Program Verification," ***IEEE Transactions on Software Engineering SE-2***, 4, 244-252.

Richardson, D.J. and L.A. Clarke (1985), "Partition Analysis: A Method Combining Testing and Verification," ***IEEE Transactions on Software Engineering SE-11,*** 12 (Dec.), 1477-1490.

Robinson, L. and K.N. Levitt (1977), "Proof Techniques for Heirarchically Structured Programs," In ***Current Trends in Programming Methodology, Vol. 2,*** R. Yeh (editor), Prentice-Hall, Englewood Cliffs, N.J., pp. 173-196.

Rosen, L.H. (1979), "Verification and Validation of a Broadcase Communications Model," ***Proceedings of the Summer Simulation Conference,*** Toronto, Canada, pp. 369-390.

Ryan, J.P. and B.C. Hodges, "A System for Automatic Evaluation of Simulation Software," In ***Proceedings of the 1976 Summer Computer Simulation Conference,*** Washington, D.C., pp. 711-717.

Saib, S.H. (1978), "SQLAB - Tools for Program Verification," In ***Tools for Embedded Computing systems Software,*** NASA Conference Publication 2064, pp. 117-120.

Saib, S.H. (1978a), "Verifiable Pascal," In ***Tools for Embedded Computing Systems Software,*** NASA Conference Publication 2064, pp. 33-36.

Saib, S.H., J.P. Benson, and R.A. Melton (1977), "A Methodology for Program Verification," ***Proceedings of the Summer Computer Simulation Conference,*** Chicago, Ill., pp. 713-720.

Sarikaya, B., G.V. Bochmann, and E. Cerny (1987), "A Test Design Methodology for Protocol Testing," ***IEEE Transactions on Software Engineering SE-13,*** 5 (May), 518-531.

Satterthwaite, E. (1972), "Debuggine Tools for High Level Languages," ***Software - Practice and Experience 2,*** 3, 197-217.

Schneidewind, N.F. (1977), "The Use of Simulation in the Evaluation of Software," ***Computer 10,*** 4, 47-53.

Smith, M.K. and R.M. Cohen (1985), "Gypsy Verification Environment: Status," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 5-6.

Sneed, H.M. and A. Merey (1985), "Automated Software Quality Assurance," ***IEEE Transactions on Software Engineering SE-11,*** 9 (Sept.), 909-916.

Steffen, J.L. (1984), "Experience with a Portable Debugging Tool," ***Software - Practice and Experience 14,*** 4, 323-334.

Stoy, J.E. (1977), ***Denotational Semantics: The Scott-Strachy Approach to Programming Language Theory***, MIT Press, Cambridge, Mass.

Stucki, L.G. (1977), "New Directions in Automated Tools for Improving Software Quality," In ***Current Trends in Programming Methodology, Vol. 2,*** R. Yeh (editor), Prentice-Hall, Englewood Cliffs, N.J., pp. 80-111.

Stucki, L.G. and G.L. Foshee (1975), "New Assertion Concepts for Self-Metric Software Validation," ***Proceedings of the International Conference on Reliable Software,*** Los Angeles, Calif., pp. 59-71.

Taylor, R.N. (1980), "Assertions in Programming Languages," ***SIGPLAN Notices 15,*** 1, 105-114.

Taylor, R.N. (1983), "Analysis of Concurrent Software by Cooperative Application of Static and Dynamic Techniques," In ***Software Validation: Inspection, Testing, Verification, Alternatives,*** Proceedings of the Symposium on Software Validation (Darmstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen (Ed.), pp. 127-137.

Tratner, M. (1979), "A Fundamental Approach to Debugging," ***Software - Practice and Experience 9,*** 97-99.

Van Horn, R.L. (1971), "Validation of Simulation Results," ***Management Science 17,*** 5 (May), 247-258.

Varga, R.J. (1978), "Prove - A Tool for Software Verification," In ***Tools for Embedded Computing Systems Software,*** NASA Conference Publication 2064, pp. 57-64.

Westley, A.E. (1979), ***Infotech State of the Art Report: Software Testing, Volume 1: Analysis and Bibliography***, Infotech International Limited, Maidenhead, Berkshire, England.

Whitehurst, R.A. (1985), "The Need for an Integrated Design, Implementation, Verification and Testing Methodology," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 97-100.

Wichmann, B.A. (1987), "Testing, validation and certification of software," ***Computer Bulletin 3,*** 4 (Dec.), 24-27.

Williams, J. (1985), "Components of Verification Technology," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 49-50.

Williams, J. and C. Applebaum (1985), "The Practical Verification System Project," ***ACM SigSoft - Software Engineering Notes 10,*** 4 (Aug.), 44-47.

Winter, E.M., D.P. Wisemiller, and J.K. Ujihara (1976), "Verification and Validation of Engineering Simulations with Minimal Data," ***Proceedings of the 1976 Summer Computer Simulation Conference*** (Washington, D.C., July 12-14). Simulation Councils, La Jolla, Calif., pp. 718-723.

Woolley, R.N. and M. Pidd (1981), "Problem Structuring — A Literature Review," ***Journal of the Operational Research Society 32***, 3 (Mar.), 197-206.

Yeh, R.T. (1977), "Verification of Programs by Predicate Transformation," In ***Current Trends in Programming Methodology, Vol. 2,*** R. Yeh (editor), Prentice-Hall, Englewood Cliffs, N.J., pp. 228-247.

Yourdon, E. (1975), ***Techniques of Program Structure and Design,*** Prentice-Hall, Englewood Cliffs, N.J.

Yourdon, E. (1979), ***Managing the Structure Techniques,*** 2nd Ed., Prentice-Hall, Englewood Cliffs, N.J.

Yourdon, E. (1982), ***Managing the System Life Cycle,*** Prentice-Hall, Englewood Cliffs, N.J.

Yourdon, E. (1985), ***Structured Walkthroughs,*** 3rd Ed., Yourdon Press, New York, N.Y.

Appendix A. Formatted Listing of SIMULATION

The formatted source listing for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The listing begins on the following page.

```
1   (*************************************************************************)
2   (*                                                                       *)
3   (*  P R O G R A M    S I M U L A T I O N                                 *)
4   (*                                                                       *)
5   (*  DESCRIPTION:  Program SIMULATION simulates a computer system that    *)
6   (*                operates in batch mode, with jobs arriving from four   *)
7   (*                different types of entry devices to a job entry        *)
8   (*                scheduler (JES).  Upon processing by the JES, a job    *)
9   (*                will be scheduled to execute on one of two CPUs, and   *)
10  (*                upon exit from the CPU, will either proceed to a       *)
11  (*                printer device (PRNT) or will leave the simulated      *)
12  (*                system to a virtual reader.  Jobs leaving the printer  *)
13  (*                go directly to the virtual reader.                     *)
14  (*                                                                       *)
15  (*                This program 'tracks' the flow of jobs through the     *)
16  (*                system using the event scheduling  world view.         *)
17  (*                Utilization of the various resources is measured, as   *)
18  (*                is the total waiting time of jobs in the system (for   *)
19  (*                computing expected waiting time).                      *)
20  (*                                                                       *)
21  (*                The simulation is repeated a specified number of       *)
22  (*                times, with each simulation consisting of a transient  *)
23  (*                state and a steady state.  Measurements are not taken  *)
24  (*                until the last transient state job leaves the system,  *)
25  (*                at which time all devices are checked for any current  *)
26  (*                use, utilization (if any) recorded, and measurement    *)
27  (*                continues until the last steady state job leaves the   *)
28  (*                system.  All measurement ceases at that point in       *)
29  (*                time.                                                  *)
30  (*                                                                       *)
31  (*  HISTORY                                                              *)
32  (*    Created By     : Richard B. Whitner                                 *)
33  (*    Date Created   : 05/05/87                                           *)
34  (*    Revised By     : Richard B. Whitner                                 *)
35  (*    Date Revised   : 12/21/87                                           *)
36  (*    Revision Notes : Program modified to improve runtime efficiency     *)
37  (*                     and to simplify the model representation.         *)
38  (*                                                                       *)
39  (*  INPUTS:   none.                                                      *)
40  (*                                                                       *)
41  (*  OUTPUTS:  Output (directed to standard output) consists of perform-  *)
42  (*            ance statistics formatted in a manner acceptable to        *)
43  (*            CIMULT, a program which will construct confidence          *)
44  (*            intervals from the generated performance measures.  The    *)
45  (*            performance measures generated consist of utilization      *)
46  (*            rate of each system facility, expected system waiting      *)
47  (*            time, and expected jobs in the system.                     *)
48  (*                                                                       *)
```

```
49      (*  CALLS:    procedure INITIALIZE                                    *)
50      (*            procedure WRITE_HEADING                                 *)
51      (*            procedure WRITE_RESULTS                                 *)
52      (*            procedure ARRIVAL                                       *)
53      (*            procedure PROCESS_EVENTS                                *)
54      (*                                                                    *)
55      (**********************************************************************)
56      {$D+}
57      {$T+}
58      program SIMULATION (output);
59
60      (*  uses MON;  *)                   (* allow instrumentation          *)
61
62        const
63           M300_mean         = 3200.0;    (* mean dist IAT for Modem 300 user  *)
64           M1200_mean        = 640.0;     (* mean dist IAT for Modem 1200 user *)
65           M2400_mean        = 1600.0;    (* mean dist IAT for Modem 2400 user *)
66           LAN_mean          = 266.67;    (* mean dist IAT for LAN user        *)
67           JES_mean          = 112.0;     (* mean dist process time for JES    *)
68           CPU1_mean         = 226.67;    (* mean dist process time for CPU1   *)
69           CPU2_mean         = 300.0;     (* mean dist process time for CPU2   *)
70           PRNT_mean         = 160.0;     (* mean dist process time for PRNT   *)
71           CPU1_probability  = 0.6;       (* probability of job using CPU1     *)
72           PRNT_probability  = 0.8;       (* probability of job using printer  *)
73           seed_const        = 15987;     (* used to generate a random number  *)
74           total_simulations = 1;         (* number of simulation repetitions  *)
75           transient_length  = 3000;      (* number of jobs in transient state *)
76           steady_state_len  = 15000;     (* number of jobs in steady state    *)
77
78        type
79           time              = real;                (* simulation time typedef *)
80           input_device_type = (M300,M1200,M2400,LAN);  (* input device types   *)
81
82           event_type = (JES_arrival,CPU1_arrival,CPU2_arrival,PRNT_arrival,
83                         SYSTEM_departure); (* enumerated event types          *)
84
85           event_ptr    = ^event_record;    (* pointer to the event list record *)
86           event_record = record
87              submit_time,                  (* job submit time                 *)
88              next_event_time : time;       (* sched time to execute next event *)
89              next_event      : event_type; (* next event to perform           *)
90              job_origin      : input_device_type; (* device job came from     *)
91              next_link       : event_ptr   (* pointer to next event in list   *)
92           end;
93
94           facility_record = record         (* data record for each facility   *)
95              time_facility_available,      (* next time facility is idle      *)
96              total_utilization_time,       (* accumulated utilization time    *)
```

```
97              mean_process_time      : time (* mean distributed process time    *)
98            end;
99
100           facility_type = array[JES_arrival..PRNT_arrival] of facility_record;
101           (* events with an associated facility in this simulation *)
102
103           input_devices = array[M300..LAN] of time;    (* enumerated array     *)
104
105           system_description = record      (* structure to hold system context *)
106             in_steady_state    : boolean; (* steady state start flag          *)
107             current_jobs,                 (* number of jobs in system now     *)
108             completed_jobs,               (* number jobs having left system   *)
109             jobs_time_j_minus_1,          (* #jobs, as of last arrival/depart *)
110             seed               : integer; (* random number generator seed     *)
111             clock,                        (* system simulation clock          *)
112             steady_state_start,           (* time the steady state began      *)
113             time_j_minus_1,               (* time of the last arrival/depart  *)
114             area,                         (* jobs-in-sys / time unit curve    *)
115             accumulated_time   : time;    (* total job waiting time           *)
116             input_device_IAT   : input_devices; (* an IAT for each job source *)
117             facility           : facility_type  (* device information        *)
118           end;
119
120       var
121         simulations : integer;            (* count of number of simulations  *)
122         system_data : system_description; (* all pertinent system data       *)
123         event_list  : event_ptr;          (* current system event list       *)
124         device      : input_device_type;  (* enumerated counter for loops    *)
125
126
127     (*************************************************************************)
128     (*                                                                       *)
129     (*  P R O C E D U R E    I N I T I A L I Z E                             *)
130     (*                                                                       *)
131     (*  DESCRIPTION:  Procedure INITIALIZE sets all system variables to their *)
132     (*                appropriate starting values.                           *)
133     (*                                                                       *)
134     (*  Procedure Calls : none.                                              *)
135     (*  Function Calls  : none.                                              *)
136     (*  Called By       : program SIMULATION.                                *)
137     (*  Parameters      : system_data -- contains pertinent system data      *)
138     (*                    event_list  -- system event list                   *)
139     (*                                                                       *)
140     (*************************************************************************)
141
142       procedure INITIALIZE(var system_data : system_description;
143                            var event_list  : event_ptr);
144
```

```
145         var
146           event_counter : event_type;            (* enumerated loop counter *)
147
148         begin
149           event_list := nil;
150           with system_data do begin
151             in_steady_state     := false;
152             current_jobs        := 0;
153             completed_jobs      := 0;
154             jobs_time_j_minus_1 := 0;
155             clock               := 0.0;
156             steady_state_start  := 0.0;
157             time_j_minus_1      := 0.0;
158             area                := 0.0;
159             accumulated_time    := 0.0;
160             for event_counter := JES_arrival to PRNT_arrival do
161               with facility[event_counter] do begin
162                 time_facility_available := 0.0;
163                 total_utilization_time  := 0.0;
164               end;
165             facility[ JES_arrival].mean_process_time := JES_mean;
166             facility[CPU1_arrival].mean_process_time := CPU1_mean;
167             facility[CPU2_arrival].mean_process_time := CPU2_mean;
168             facility[PRNT_arrival].mean_process_time := PRNT_mean;
169             input_device_IAT[ M300]                  := M300_mean;
170             input_device_IAT[M1200]                  := M1200_mean;
171             input_device_IAT[M2400]                  := M2400_mean;
172             input_device_IAT[  LAN]                  := LAN_mean;
173           end;
174         end;  (* procedure INITIALIZE *)
175
176
177       (*****************************************************************************)
178       (*                                                                           *)
179       (*  F U N C T I O N    R A N D                                               *)
180       (*                                                                           *)
181       (*  DESCRIPTION:  Function RAND returns a pseudo-random number N between     *)
182       (*                0 (inclusive) and 1, based on the seed parameter.          *)
183       (*                                                                           *)
184       (*  Procedure Calls : none.                                                  *)
185       (*  Function Calls  : none.                                                  *)
186       (*  Called By       : function EXPON.                                        *)
187       (*                    procedure PROCESS_EVENT.                               *)
188       (*  Parameters      : seed -- random number generator seed value.            *)
189       (*                                                                           *)
190       (*****************************************************************************)
191
192         function RAND(var seed : integer) : time;
```

```
193
194          const
195             d2p31m = 2147483647.0;
196
197          var
198             quotient,
199             z         : time;
200             intquot   : integer;
201
202          begin
203             z        := seed;
204             z        := 16807.0 * z;
205             quotient := z / d2p31m;
206             intquot  := trunc(quotient);
207             z        := z - intquot * d2p31m;
208             seed     := trunc(z);
209          {  RAND     := z /d2p31m  }
210             RAND     := RANDOM
211          end;  (* function RAND *)
212
213
214       (*****************************************************************************)
215       (*                                                                           *)
216       (*  F U N C T I O N    E X P O N                                             *)
217       (*                                                                           *)
218       (*  DESCRIPTION:  Function EXPON generates exponentially distributed         *)
219       (*                values based on a given mean.                              *)
220       (*                                                                           *)
221       (*  Procedure Calls : none.                                                  *)
222       (*  Function Calls  : RAND.                                                  *)
223       (*  Called By       : procedure ARRIVAL.                                     *)
224       (*                    procedure SCHEDULE.                                    *)
225       (*  Parameters      : mean -- mean distribution value                        *)
226       (*                    seed -- random number generator seed value             *)
227       (*                                                                           *)
228       (*****************************************************************************)
229
230         function EXPON(mean : time;var seed : integer) : time;
231
232           var
233              r, ex : time;
234
235          begin
236             EXPON := -mean * LN(RAND(seed))
237          end;  (* function EXPON *)
238
239
240       (*****************************************************************************)
```

```
241     (*                                                                    *)
242     (*  P R O C E D U R E    A R E A _ J                                  *)
243     (*                                                                    *)
244     (*  DESCRIPTION:  Procedure AREA_J computes the area beneath the curve *)
245     (*                given by the number of jobs in the system at time t.*)
246     (*                In this particular instance, the area computed is that *)
247     (*                of all jobs in the system since the last computation *)
248     (*                of the area (i.e., whenever there is an arrival to or *)
249     (*                departure from the system).                         *)
250     (*                                                                    *)
251     (*  Procedure Calls : none.                                           *)
252     (*  Function Calls  : none.                                           *)
253     (*  Called By       : procedure ARRIVAL.                              *)
254     (*                    procedure DEPARTURE.                            *)
255     (*  Parameters      : system_data -- pertinent system data            *)
256     (*                                                                    *)
257     (**********************************************************************)
258
259       procedure AREA_J(var system_data : system_description);
260
261 ┌──── begin
262 │┌─── with system_data do begin
263 ││       area := area + jobs_time_j_minus_1 * (clock - time_j_minus_1);
264 ││       jobs_time_j_minus_1 := current_jobs;
265 ││       time_j_minus_1      := clock
266 │└─── end
267 └──── end;  (* procedure AREA_J *)
268
269
270     (**********************************************************************)
271     (*                                                                    *)
272     (*  P R O C E D U R E    I N S E R T                                  *)
273     (*                                                                    *)
274     (*  DESCRIPTION:  Procedure INSERT places an event in the proper position *)
275     (*                in the system event list, in order of ascending time. *)
276     (*                INSERT compares the time of the item to be inserted *)
277     (*                against the next item in the event list and calls   *)
278     (*                itself recursively as needed until the proper location *)
279     (*                is found.                                           *)
280     (*                                                                    *)
281     (*  Procedure Calls : INSERT (recursively)                            *)
282     (*  Function Calls  : none.                                           *)
283     (*  Called By       : procedure INSERT (recursively)                  *)
284     (*                    procedure ARRIVAL.                              *)
285     (*                    procedure PROCESS_EVENT.                        *)
286     (*  Parameters      : event      -- the event to be inserted          *)
287     (*                    event_list -- the current events list           *)
288     (*                                                                    *)
```

```
289     (*****************************************************************************)
290
291       procedure INSERT(var event,event_list : event_ptr);
292
293     ┌──── begin
294     │ ┌──── if event_list = nil then begin    (* list empty, insert at the top *)
295     │ │       event^.next_link := event_list;
296     │ │       event_list       := event
297     │ └──── end
298     │       else
299     │         if event^.next_event_time >= event_list^.next_event_time then
300     │           INSERT(event,event_list^.next_link)
301     │ ┌──── else begin
302     │ │       event^.next_link := event_list;
303     │ │       event_list       := event
304     │ └──── end
305     └──── end;  (* procedure INSERT *)
306
307
308     (*****************************************************************************)
309     (*                                                                           *)
310     (*  P R O C E D U R E    A R R I V A L                                       *)
311     (*                                                                           *)
312     (*  DESCRIPTION:  Procedure ARRIVAL generates an arrival event from the      *)
313     (*                same type of device that generated the job now calling     *)
314     (*                ARRIVAL.  The submit time is recorded, other necessary     *)
315     (*                values set, and the new event inserted into the event      *)
316     (*                list.                                                      *)
317     (*                                                                           *)
318     (*  Procedure Calls : INSERT                                                 *)
319     (*  Function Calls  : none.                                                  *)
320     (*  Called By       : program SIMULATION.                                    *)
321     (*                    procedure PROCESS_EVENT.                               *)
322     (*  Parameters      : origin      -- type of device generating an arrival    *)
323     (*                    event_list  -- the current events list                 *)
324     (*                    system_data -- pertinent system data                   *)
325     (*                                                                           *)
326     (*****************************************************************************)
327
328       procedure ARRIVAL(origin : input_device_type;var event_list : event_ptr;
329                         var system_data : system_description);
330
331         var
332           event : event_ptr;        (* a new event to be added to the system *)
333           mean  : time;             (* mean distributed IAT time for device  *)
334
335     ┌──── begin
336     │       NEW(event);
```

```
337           with event^ do
338   ┌─────── with system_data do begin
339   │           job_origin      := origin;
340   │           mean            := input_device_IAT[origin];
341   │           submit_time     := EXPON(mean,seed) + clock;
342   │           next_event_time := submit_time;
343   │           next_event      := JES_arrival;
344   │           if in_steady_state then
345   │              AREA_J(system_data)
346   └─────── end;
347           INSERT(event,event_list)
348   └─────  end;  (* procedure ARRIVAL *)
349
350
351       (*************************************************************************)
352       (*                                                                       *)
353       (*  P R O C E D U R E    P R O C E S S _ E V E N T                       *)
354       (*                                                                       *)
355       (*  DESCRIPTION:  Procedure PROCESS_EVENT determines which type of event  *)
356       (*                occurs next for a given job and then SCHEDULEs that    *)
357       (*                event and INSERTs the job back into the current events *)
358       (*                list, provided the next event is not a system departure *)
359       (*                event.                                                 *)
360       (*                                                                       *)
361       (*  Procedure Calls : ARRIVAL.                                           *)
362       (*                    DEPARTURE.                                         *)
363       (*                    INSERT.                                            *)
364       (*                    SCHEDULE.                                          *)
365       (*  Function Calls  : RAND.                                              *)
366       (*  Called By       : program SIMULATION.                                *)
367       (*  Parameters      : event_list  -- the current events list             *)
368       (*                    system_data -- pertinent system data               *)
369       (*                                                                       *)
370       (*************************************************************************)
371
372         procedure PROCESS_EVENT(var event_list  : event_ptr;
373                                 var system_data : system_description);
374
375           var
376              this_event : event_ptr;                    (* the current event *)
377
378
379       (*************************************************************************)
380       (*                                                                       *)
381       (*  P R O C E D U R E    S C H E D U L E                                 *)
382       (*                                                                       *)
383       (*  DESCRIPTION:  Procedure SCHEDULE is a generic scheduling routine     *)
384       (*                which performs the scheduling tasks common to all      *)
```

```
385       (*              events.  It determines if the event scheduled to occur  *)
386       (*              can take place at the scheduled time, and if it can't,   *)
387       (*              it determines when it could occur.  In either case,      *)
388       (*              the length of the event is determined, and then the      *)
389       (*              next event is scheduled, all of this relative to when    *)
390       (*              the event actually occurred.                             *)
391       (*                                                                       *)
392       (*  Procedure Calls : none.                                              *)
393       (*  Function Calls  : EXPON.                                             *)
394       (*  Called By       : program SIMULATION.                                *)
395       (*                    procedure PROCESS_EVENT.                           *)
396       (*  Parameters      : event       -- the event to be scheduled           *)
397       (*                    system_data -- pertinent system data               *)
398       (*                                                                       *)
399       (*************************************************************************)
400
401          procedure SCHEDULE(var event : event_ptr;
402                             var system_data : system_description);
403
404          var
405             length_of_event : time;    (* the amount of time to complete event *)
406
407          begin
408             with event^ do
409                with system_data do
410                with facility[next_event] do begin
411                   length_of_event := EXPON(mean_process_time,seed);
412                   if time_facility_available <= next_event_time then
413                      time_facility_available := next_event_time +
                         length_of_event
414                   else
415                      time_facility_available := time_facility_available +
416                                                 length_of_event;
417                   if in_steady_state then
418                      total_utilization_time := total_utilization_time +
419                                                length_of_event;
420                   next_event_time := time_facility_available
421                end
422          end;  (* procedure SCHEDULE *)
423
424
425       (*************************************************************************)
426       (*                                                                       *)
427       (*  P R O C E D U R E    D E P A R T U R E                               *)
428       (*                                                                       *)
429       (*  DESCRIPTION:  Procedure DEPARTURE handles the exiting of a job from  *)
430       (*                the system, checking to see whether or not the system  *)
431       (*                has entered the steady state.  As soon as the system   *)
```

```
432       (*              enters the steady state, all statistical values are     *)
433       (*              initialized and the system flagged as being in the      *)
434       (*              steady state.                                           *)
435       (*                                                                      *)
436       (*  Procedure Calls : AREA_J.                                           *)
437       (*  Function Calls  : none.                                             *)
438       (*  Called By       : procedure PROCESS_EVENT.                          *)
439       (*  Parameters      : event       -- the departing event                *)
440       (*                    system_data -- pertinent system data              *)
441       (*                                                                      *)
442       (************************************************************************)
443
444          procedure DEPARTURE(var event : event_ptr;
445                              var system_data : system_description);
446
447          begin
448             with event^ do
449             with system_data do begin
450                current_jobs   := current_jobs - 1;
451                completed_jobs := completed_jobs + 1;
452             if in_steady_state then begin
453                AREA_J(system_data);
454                accumulated_time := accumulated_time + (next_event_time -
455                                    submit_time)
456             end
457             else
458             if completed_jobs = transient_length then begin
459                in_steady_state     := true;
460                steady_state_start  := clock;
461                time_j_minus_1      := clock;
462                jobs_time_j_minus_1 := current_jobs;
463                completed_jobs      := 0
464             end
465          end;
466          DISPOSE(event)
467       end;  (* procedure DEPARTURE *)
468
469
470       begin (* procedure PROCESS_EVENT *)
471          this_event := event_list;
472          event_list := event_list^.next_link;
473          with this_event^ do
474          with system_data do begin
475             clock := next_event_time;              (* advance system clock *)
476          if this_event^.next_event <> SYSTEM_departure then begin
477             SCHEDULE(this_event,system_data);
478          case next_event of
479             JES_arrival : begin
```

```
480                  current_jobs := current_jobs + 1;
481                  ARRIVAL(job_origin,event_list,system_data);
482                  if RAND(seed) < CPU1_probability then
483                     next_event := CPU1_arrival
484                  else
485                     next_event := CPU2_arrival
486               end;
487               CPU1_arrival,CPU2_arrival :
488                  if RAND(seed) < PRNT_probability then
489                     next_event := PRNT_arrival
490                  else
491                     next_event := SYSTEM_departure;
492               PRNT_arrival :
493                  next_event := SYSTEM_departure
494            end;
495            INSERT(this_event,event_list)
496         end
497         else
498            DEPARTURE(this_event,system_data);
499      end
500   end;  (* procedure PROCESS_EVENT *)
501
502
503   (*****************************************************************************)
504   (*                                                                           *)
505   (*  P R O C E D U R E    W R I T E _ H E A D I N G                           *)
506   (*                                                                           *)
507   (*  DESCRIPTION:  Procedure WRITE_HEADING writes the output heading in       *)
508   (*                a form which can be utilized by CIMULT.                    *)
509   (*                                                                           *)
510   (*  Procedure Calls : none.                                                  *)
511   (*  Function Calls  : none.                                                  *)
512   (*  Called By       : program SIMULATION.                                    *)
513   (*  Parameters      : none.                                                  *)
514   (*                                                                           *)
515   (*****************************************************************************)
516
517      procedure WRITE_HEADING;
518
519         const
520            output_items = 6;
521
522      begin
523         writeln(output_items:2);
524         writeln('Utilization of the Job Entry Subsystem Scheduler');
525         writeln('Utilization of CPU 1');
526         writeln('Utilization of CPU 2');
527         writeln('Utilization of the Printer');
```

```
528             writeln('Expected Waiting Time');
529             writeln('Expected Jobs in the System')
530         end;  (* procedure WRITE_HEADING *)
531
532
533     (*************************************************************************)
534     (*                                                                       *)
535     (*  P R O C E D U R E    W R I T E _ R E S U L T S                       *)
536     (*                                                                       *)
537     (*  DESCRIPTION:  Procedure WRITE_RESULTS writes the results of the      *)
538     (*                simulation in a form which can be utilized by CIMULT.  *)
539     (*                Using the accumulated system statistics, all appro-    *)
540     (*                priate values are calculated before any output is      *)
541     (*                generated.                                             *)
542     (*                                                                       *)
543     (*  Procedure Calls : none.                                              *)
544     (*  Function Calls  : none.                                              *)
545     (*  Called By       : program SIMULATION.                                *)
546     (*  Parameters      : statistics -- system statistical totals            *)
547     (*                                                                       *)
548     (*************************************************************************)
549
550     procedure WRITE_RESULTS(statistics : system_description);
551
552       var
553         JES_util,       (* Job Entry Scheduler utilization time           *)
554         CPU1_util,      (* CPU1 utilization time                          *)
555         CPU2_util,      (* CPU2 utilization time                          *)
556         PRNT_util,      (* Printer utilization time                       *)
557         exp_jobs,       (* Expected (avg.) number of jobs at any given time *)
558         exp_time,       (* Expected (avg.) waiting time for any given job  *)
559         steady_time : time;           (* system time in the steady state  *)
560
561     begin
562       with statistics do begin
563         steady_time := clock - steady_state_start;
564         JES_util  := facility[ JES_arrival].total_utilization_time /
            steady_time;
565         CPU1_util := facility[CPU1_arrival].total_utilization_time /
            steady_time;
566         CPU2_util := facility[CPU2_arrival].total_utilization_time /
            steady_time;
567         PRNT_util := facility[PRNT_arrival].total_utilization_time /
            steady_time;
568         exp_time  := area / steady_state_len;
569         exp_jobs  := area / steady_time
570       end;
571       write(JES_util:8:5);
```

```
572         write(CPU1_util:13:5);
573         write(CPU2_util:13:5);
574         write(PRNT_util:13:5);
575         write(exp_time:17:5);
576         writeln(exp_jobs:15:5)
577     end;  (* procedure WRITE_RESULTS *)
578
579
580     (*****************************************************************************)
581     (*              M A I N      P R O G R A M      D R I V E R                   *)
582     (*****************************************************************************)
583     BEGIN  (* program SIMULATION *)
584       WRITE_HEADING;
585       for simulations := 1 to total_simulations do begin
586         with system_data do
587           seed := simulations * seed_const + (simulations mod 2) + 1;
588         INITIALIZE(system_data,event_list);
589
590         (************************************************)
591         (* Schedule the first job from each entry device *)
592         (************************************************)
593         for device := M300 to LAN do
594           ARRIVAL(device,event_list,system_data);
595
596         (************************************************)
597         (* Process the specified number of transient and *)
598         (* steady-state jobs.                            *)
599         (************************************************)
600         while system_data.completed_jobs < steady_state_len do
601           PROCESS_EVENT(event_list,system_data);
602
603         WRITE_RESULTS(system_data)
604       end
605     END.
```

Appendix B. Cross-Reference Report

The cross-reference report for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The report begins on the following page.

```
                    Cross-Reference Report for
                  D:\RICK\THESIS\EVENTSCH.PAS
                  Saturday 9/10/88 at 13:03:51
```

```
accumulated_time : time (Field)
  dec in EVENTSCH.PAS  115 SIMULATION.system_description.
  set in EVENTSCH.PAS  159 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  454 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  454 SIMULATION.PROCESS_EVENT\DEPARTURE\

area : time (Field)
  dec in EVENTSCH.PAS  114 SIMULATION.system_description.
  set in EVENTSCH.PAS  158 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  263 SIMULATION.AREA_J\
  ref in EVENTSCH.PAS  263 SIMULATION.AREA_J\
  ref in EVENTSCH.PAS  568 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  569 SIMULATION.WRITE_RESULTS\

AREA_J (Proc)
  dec in EVENTSCH.PAS  259 SIMULATION.
  ref in EVENTSCH.PAS  345 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  453 SIMULATION.PROCESS_EVENT\DEPARTURE\

ARRIVAL (Proc)
  dec in EVENTSCH.PAS  328 SIMULATION.
  ref in EVENTSCH.PAS  481 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  594 SIMULATION.

clock : time (Field)
  dec in EVENTSCH.PAS  111 SIMULATION.system_description.
  set in EVENTSCH.PAS  155 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  263 SIMULATION.AREA_J\
  ref in EVENTSCH.PAS  266 SIMULATION.AREA_J\
  ref in EVENTSCH.PAS  341 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  460 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  461 SIMULATION.PROCESS_EVENT\DEPARTURE\
  set in EVENTSCH.PAS  475 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  563 SIMULATION.WRITE_RESULTS\

completed_jobs : Integer (Field)
  dec in EVENTSCH.PAS  108 SIMULATION.system_description.
  set in EVENTSCH.PAS  153 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  451 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  451 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  458 SIMULATION.PROCESS_EVENT\DEPARTURE\
  set in EVENTSCH.PAS  463 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  600 SIMULATION.
```

```
CPU1_arrival (Const)
  dec in EVENTSCH.PAS   82 SIMULATION.
  ref in EVENTSCH.PAS  166 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  484 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  487 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  565 SIMULATION.WRITE_RESULTS\

CPU1_mean (Const)
  dec in EVENTSCH.PAS   68 SIMULATION.
  ref in EVENTSCH.PAS  166 SIMULATION.INITIALIZE\

CPU1_probability (Const)
  dec in EVENTSCH.PAS   71 SIMULATION.
  ref in EVENTSCH.PAS  482 SIMULATION.PROCESS_EVENT\

CPU1_util : time (Var)
  dec in EVENTSCH.PAS  554 SIMULATION.WRITE_RESULTS\
  set in EVENTSCH.PAS  565 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  572 SIMULATION.WRITE_RESULTS\

CPU2_arrival (Const)
  dec in EVENTSCH.PAS   82 SIMULATION.
  ref in EVENTSCH.PAS  167 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  486 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  487 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  566 SIMULATION.WRITE_RESULTS\

CPU2_mean (Const)
  dec in EVENTSCH.PAS   69 SIMULATION.
  ref in EVENTSCH.PAS  167 SIMULATION.INITIALIZE\

CPU2_util : time (Var)
  dec in EVENTSCH.PAS  555 SIMULATION.WRITE_RESULTS\
  set in EVENTSCH.PAS  566 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  573 SIMULATION.WRITE_RESULTS\

current_jobs : Integer (Field)
  dec in EVENTSCH.PAS  107 SIMULATION.system_description.
  set in EVENTSCH.PAS  152 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  264 SIMULATION.AREA_J\
  set in EVENTSCH.PAS  450 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  450 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  462 SIMULATION.PROCESS_EVENT\DEPARTURE\
  set in EVENTSCH.PAS  480 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  480 SIMULATION.PROCESS_EVENT\

d2p31m (Const)
  dec in EVENTSCH.PAS  195 SIMULATION.RAND\
  ref in EVENTSCH.PAS  205 SIMULATION.RAND\
  ref in EVENTSCH.PAS  207 SIMULATION.RAND\

DEPARTURE (Proc)
```

```
  dec in EVENTSCH.PAS  444 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  498 SIMULATION.PROCESS_EVENT\

device : input_device_type (Var)
  dec in EVENTSCH.PAS  124 SIMULATION.
  set in EVENTSCH.PAS  593 SIMULATION.
  ref in EVENTSCH.PAS  594 SIMULATION.

event : event_ptr (Var)
  dec in EVENTSCH.PAS  332 SIMULATION.ARRIVAL\
  var in EVENTSCH.PAS  336 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  337 SIMULATION.ARRIVAL\
  var in EVENTSCH.PAS  347 SIMULATION.ARRIVAL\

event : event_ptr (VarParam)
  dec in EVENTSCH.PAS  291 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  295 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  297 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  299 SIMULATION.INSERT\
  var in EVENTSCH.PAS  300 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  302 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  304 SIMULATION.INSERT\

event : event_ptr (VarParam)
  dec in EVENTSCH.PAS  444 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  448 SIMULATION.PROCESS_EVENT\DEPARTURE\
  var in EVENTSCH.PAS  466 SIMULATION.PROCESS_EVENT\DEPARTURE\

event : event_ptr (VarParam)
  dec in EVENTSCH.PAS  401 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  408 SIMULATION.PROCESS_EVENT\SCHEDULE\

event_counter : event_type (Var)
  dec in EVENTSCH.PAS  146 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  160 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  161 SIMULATION.INITIALIZE\

event_list : event_ptr (Var)
  dec in EVENTSCH.PAS  123 SIMULATION.
  var in EVENTSCH.PAS  588 SIMULATION.
  var in EVENTSCH.PAS  594 SIMULATION.
  var in EVENTSCH.PAS  601 SIMULATION.

event_list : event_ptr (VarParam)
  dec in EVENTSCH.PAS  328 SIMULATION.ARRIVAL\
  var in EVENTSCH.PAS  347 SIMULATION.ARRIVAL\

event_list : event_ptr (VarParam)
  dec in EVENTSCH.PAS  143 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  149 SIMULATION.INITIALIZE\

event_list : event_ptr (VarParam)
```

```
  dec in EVENTSCH.PAS  291 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  294 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  295 SIMULATION.INSERT\
  set in EVENTSCH.PAS  296 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  299 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  300 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  302 SIMULATION.INSERT\
  set in EVENTSCH.PAS  303 SIMULATION.INSERT\

event_list : event_ptr (VarParam)
  dec in EVENTSCH.PAS  372 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  471 SIMULATION.PROCESS_EVENT\
  set in EVENTSCH.PAS  472 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  472 SIMULATION.PROCESS_EVENT\
  var in EVENTSCH.PAS  481 SIMULATION.PROCESS_EVENT\
  var in EVENTSCH.PAS  495 SIMULATION.PROCESS_EVENT\

event_ptr = ^event_record (Type)
  dec in EVENTSCH.PAS   85 SIMULATION.
  ref in EVENTSCH.PAS   91 SIMULATION.event_record.
  ref in EVENTSCH.PAS  123 SIMULATION.
  ref in EVENTSCH.PAS  143 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  291 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  328 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  332 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  372 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  376 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  401 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  444 SIMULATION.PROCESS_EVENT\DEPARTURE\

event_record = Record (Type)
  dec in EVENTSCH.PAS   85 SIMULATION.
  ref in EVENTSCH.PAS   86 SIMULATION.

event_type = Enumerated (Type)
  dec in EVENTSCH.PAS   82 SIMULATION.
  ref in EVENTSCH.PAS   89 SIMULATION.event_record.
  ref in EVENTSCH.PAS  146 SIMULATION.INITIALIZE\

ex : time (Var)
  dec in EVENTSCH.PAS  233 SIMULATION.EXPON\

EXPON : time (Function)
  dec in EVENTSCH.PAS  230 SIMULATION.
  set in EVENTSCH.PAS  236 SIMULATION.EXPON\
  ref in EVENTSCH.PAS  341 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  411 SIMULATION.PROCESS_EVENT\SCHEDULE\

exp_jobs : time (Var)
  dec in EVENTSCH.PAS  557 SIMULATION.WRITE_RESULTS\
  set in EVENTSCH.PAS  569 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  576 SIMULATION.WRITE_RESULTS\
```

```
exp_time : time (Var)
  dec in EVENTSCH.PAS  558 SIMULATION.WRITE_RESULTS\
  set in EVENTSCH.PAS  568 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  575 SIMULATION.WRITE_RESULTS\

facility : facility_type (Field)
  dec in EVENTSCH.PAS  117 SIMULATION.system_description.
  ref in EVENTSCH.PAS  161 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  165 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  166 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  167 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  168 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  410 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  564 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  565 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  566 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  567 SIMULATION.WRITE_RESULTS\

facility_record = Record (Type)
  dec in EVENTSCH.PAS   94 SIMULATION.
  ref in EVENTSCH.PAS  100 SIMULATION.

facility_type = Array[Subrange] of facility_record (Type)
  dec in EVENTSCH.PAS  100 SIMULATION.
  ref in EVENTSCH.PAS  117 SIMULATION.system_description.

INITIALIZE (Proc)
  dec in EVENTSCH.PAS  142 SIMULATION.
  ref in EVENTSCH.PAS  588 SIMULATION.

input_devices = Array[Subrange] of time (Type)
  dec in EVENTSCH.PAS  103 SIMULATION.
  ref in EVENTSCH.PAS  116 SIMULATION.system_description.

input_device_IAT : input_devices (Field)
  dec in EVENTSCH.PAS  116 SIMULATION.system_description.
  set in EVENTSCH.PAS  169 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  170 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  171 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  172 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  340 SIMULATION.ARRIVAL\

input_device_type = Enumerated (Type)
  dec in EVENTSCH.PAS   80 SIMULATION.
  ref in EVENTSCH.PAS   90 SIMULATION.event_record.
  ref in EVENTSCH.PAS  124 SIMULATION.
  ref in EVENTSCH.PAS  328 SIMULATION.ARRIVAL\

INSERT (Proc)
  dec in EVENTSCH.PAS  291 SIMULATION.
  ref in EVENTSCH.PAS  300 SIMULATION.INSERT\
```

```
  ref in EVENTSCH.PAS  347 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  495 SIMULATION.PROCESS_EVENT\

intquot : Integer (Var)
  dec in EVENTSCH.PAS  200 SIMULATION.RAND\
  set in EVENTSCH.PAS  206 SIMULATION.RAND\
  ref in EVENTSCH.PAS  207 SIMULATION.RAND\

in_steady_state : Boolean (Field)
  dec in EVENTSCH.PAS  106 SIMULATION.system_description.
  set in EVENTSCH.PAS  151 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  344 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  417 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  452 SIMULATION.PROCESS_EVENT\DEPARTURE\
  set in EVENTSCH.PAS  459 SIMULATION.PROCESS_EVENT\DEPARTURE\

JES_arrival (Const)
  dec in EVENTSCH.PAS   82 SIMULATION.
  ref in EVENTSCH.PAS  100 SIMULATION.
  ref in EVENTSCH.PAS  160 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  165 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  343 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  479 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  564 SIMULATION.WRITE_RESULTS\

JES_mean (Const)
  dec in EVENTSCH.PAS   67 SIMULATION.
  ref in EVENTSCH.PAS  165 SIMULATION.INITIALIZE\

JES_util : time (Var)
  dec in EVENTSCH.PAS  553 SIMULATION.WRITE_RESULTS\
  set in EVENTSCH.PAS  564 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  571 SIMULATION.WRITE_RESULTS\

jobs_time_j_minus_1 : Integer (Field)
  dec in EVENTSCH.PAS  109 SIMULATION.system_description.
  set in EVENTSCH.PAS  154 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  263 SIMULATION.AREA_J\
  set in EVENTSCH.PAS  264 SIMULATION.AREA_J\
  set in EVENTSCH.PAS  462 SIMULATION.PROCESS_EVENT\DEPARTURE\

job_origin : input_device_type (Field)
  dec in EVENTSCH.PAS   90 SIMULATION.event_record.
  set in EVENTSCH.PAS  339 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  481 SIMULATION.PROCESS_EVENT\

LAN (Const)
  dec in EVENTSCH.PAS   80 SIMULATION.
  ref in EVENTSCH.PAS  103 SIMULATION.
  ref in EVENTSCH.PAS  172 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  593 SIMULATION.
```

```
LAN_mean (Const)
  dec in EVENTSCH.PAS   66 SIMULATION.
  ref in EVENTSCH.PAS  172 SIMULATION.INITIALIZE\

length_of_event : time (Var)
  dec in EVENTSCH.PAS  405 SIMULATION.PROCESS_EVENT\SCHEDULE\
  set in EVENTSCH.PAS  411 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  414 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  416 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  419 SIMULATION.PROCESS_EVENT\SCHEDULE\

M1200 (Const)
  dec in EVENTSCH.PAS   80 SIMULATION.
  ref in EVENTSCH.PAS  170 SIMULATION.INITIALIZE\

M1200_mean (Const)
  dec in EVENTSCH.PAS   64 SIMULATION.
  ref in EVENTSCH.PAS  170 SIMULATION.INITIALIZE\

M2400 (Const)
  dec in EVENTSCH.PAS   80 SIMULATION.
  ref in EVENTSCH.PAS  171 SIMULATION.INITIALIZE\

M2400_mean (Const)
  dec in EVENTSCH.PAS   65 SIMULATION.
  ref in EVENTSCH.PAS  171 SIMULATION.INITIALIZE\

M300 (Const)
  dec in EVENTSCH.PAS   80 SIMULATION.
  ref in EVENTSCH.PAS  103 SIMULATION.
  ref in EVENTSCH.PAS  169 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  593 SIMULATION.

M300_mean (Const)
  dec in EVENTSCH.PAS   63 SIMULATION.
  ref in EVENTSCH.PAS  169 SIMULATION.INITIALIZE\

mean : time (Var)
  dec in EVENTSCH.PAS  333 SIMULATION.ARRIVAL\
  set in EVENTSCH.PAS  340 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  341 SIMULATION.ARRIVAL\

mean : time (ValParam)
  dec in EVENTSCH.PAS  230 SIMULATION.EXPON\
  ref in EVENTSCH.PAS  236 SIMULATION.EXPON\

mean_process_time : time (Field)
  dec in EVENTSCH.PAS   97 SIMULATION.facility_record.
  set in EVENTSCH.PAS  165 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  166 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  167 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  168 SIMULATION.INITIALIZE\
```

```
  ref in EVENTSCH.PAS  411 SIMULATION.PROCESS_EVENT\SCHEDULE\

next_event : event_type (Field)
  dec in EVENTSCH.PAS   89 SIMULATION.event_record.
  set in EVENTSCH.PAS  343 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  410 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  476 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  478 SIMULATION.PROCESS_EVENT\
  set in EVENTSCH.PAS  483 SIMULATION.PROCESS_EVENT\
  set in EVENTSCH.PAS  485 SIMULATION.PROCESS_EVENT\
  set in EVENTSCH.PAS  489 SIMULATION.PROCESS_EVENT\
  set in EVENTSCH.PAS  491 SIMULATION.PROCESS_EVENT\
  set in EVENTSCH.PAS  493 SIMULATION.PROCESS_EVENT\

next_event_time : time (Field)
  dec in EVENTSCH.PAS   88 SIMULATION.event_record.
  ref in EVENTSCH.PAS  299 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  299 SIMULATION.INSERT\
  set in EVENTSCH.PAS  342 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  412 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  413 SIMULATION.PROCESS_EVENT\SCHEDULE\
  set in EVENTSCH.PAS  420 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  454 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  475 SIMULATION.PROCESS_EVENT\

next_link : event_ptr (Field)
  dec in EVENTSCH.PAS   91 SIMULATION.event_record.
  set in EVENTSCH.PAS  295 SIMULATION.INSERT\
  var in EVENTSCH.PAS  300 SIMULATION.INSERT\
  set in EVENTSCH.PAS  302 SIMULATION.INSERT\
  ref in EVENTSCH.PAS  472 SIMULATION.PROCESS_EVENT\

origin : input_device_type (ValParam)
  dec in EVENTSCH.PAS  328 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  339 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  340 SIMULATION.ARRIVAL\

output_items (Const)
  dec in EVENTSCH.PAS  520 SIMULATION.WRITE_HEADING\
  ref in EVENTSCH.PAS  523 SIMULATION.WRITE_HEADING\

PRNT_arrival (Const)
  dec in EVENTSCH.PAS   82 SIMULATION.
  ref in EVENTSCH.PAS  100 SIMULATION.
  ref in EVENTSCH.PAS  160 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  168 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  490 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  492 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  567 SIMULATION.WRITE_RESULTS\

PRNT_mean (Const)
  dec in EVENTSCH.PAS   70 SIMULATION.
```

```
  ref in EVENTSCH.PAS  168 SIMULATION.INITIALIZE\

PRNT_probability (Const)
  dec in EVENTSCH.PAS   72 SIMULATION.
  ref in EVENTSCH.PAS  488 SIMULATION.PROCESS_EVENT\

PRNT_util : time (Var)
  dec in EVENTSCH.PAS  556 SIMULATION.WRITE_RESULTS\
  set in EVENTSCH.PAS  567 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  574 SIMULATION.WRITE_RESULTS\

PROCESS_EVENT (Proc)
  dec in EVENTSCH.PAS  372 SIMULATION.
  ref in EVENTSCH.PAS  601 SIMULATION.

quotient : time (Var)
  dec in EVENTSCH.PAS  198 SIMULATION.RAND\
  set in EVENTSCH.PAS  205 SIMULATION.RAND\
  ref in EVENTSCH.PAS  206 SIMULATION.RAND\

r : time (Var)
  dec in EVENTSCH.PAS  233 SIMULATION.EXPON\

RAND : time (Function)
  dec in EVENTSCH.PAS  192 SIMULATION.
  set in EVENTSCH.PAS  210 SIMULATION.RAND\
  ref in EVENTSCH.PAS  236 SIMULATION.EXPON\
  ref in EVENTSCH.PAS  482 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  488 SIMULATION.PROCESS_EVENT\

SCHEDULE (Proc)
  dec in EVENTSCH.PAS  401 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  477 SIMULATION.PROCESS_EVENT\

seed : Integer (VarParam)
  dec in EVENTSCH.PAS  230 SIMULATION.EXPON\
  var in EVENTSCH.PAS  236 SIMULATION.EXPON\

seed : Integer (VarParam)
  dec in EVENTSCH.PAS  192 SIMULATION.RAND\
  ref in EVENTSCH.PAS  203 SIMULATION.RAND\
  set in EVENTSCH.PAS  208 SIMULATION.RAND\

seed : Integer (Field)
  dec in EVENTSCH.PAS  110 SIMULATION.system_description.
  var in EVENTSCH.PAS  341 SIMULATION.ARRIVAL\
  var in EVENTSCH.PAS  411 SIMULATION.PROCESS_EVENT\SCHEDULE\
  var in EVENTSCH.PAS  482 SIMULATION.PROCESS_EVENT\
  var in EVENTSCH.PAS  488 SIMULATION.PROCESS_EVENT\
  set in EVENTSCH.PAS  587 SIMULATION.

seed_const (Const)
```

```
  dec in EVENTSCH.PAS   73 SIMULATION.
  ref in EVENTSCH.PAS  587 SIMULATION.

SIMULATION (Program)
  dec in EVENTSCH.PAS   58 SIMULATION.

simulations : Integer (Var)
  dec in EVENTSCH.PAS  121 SIMULATION.
  set in EVENTSCH.PAS  585 SIMULATION.
  ref in EVENTSCH.PAS  587 SIMULATION.
  ref in EVENTSCH.PAS  587 SIMULATION.

statistics : system_description (ValParam)
  dec in EVENTSCH.PAS  550 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  562 SIMULATION.WRITE_RESULTS\

steady_state_len (Const)
  dec in EVENTSCH.PAS   76 SIMULATION.
  ref in EVENTSCH.PAS  568 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  600 SIMULATION.

steady_state_start : time (Field)
  dec in EVENTSCH.PAS  112 SIMULATION.system_description.
  set in EVENTSCH.PAS  156 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  460 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  563 SIMULATION.WRITE_RESULTS\

steady_time : time (Var)
  dec in EVENTSCH.PAS  559 SIMULATION.WRITE_RESULTS\
  set in EVENTSCH.PAS  563 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  564 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  565 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  566 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  567 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  570 SIMULATION.WRITE_RESULTS\

submit_time : time (Field)
  dec in EVENTSCH.PAS   87 SIMULATION.event_record.
  set in EVENTSCH.PAS  341 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  342 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  455 SIMULATION.PROCESS_EVENT\DEPARTURE\

system_data : system_description (Var)
  dec in EVENTSCH.PAS  122 SIMULATION.
  ref in EVENTSCH.PAS  586 SIMULATION.
  var in EVENTSCH.PAS  588 SIMULATION.
  var in EVENTSCH.PAS  594 SIMULATION.
  ref in EVENTSCH.PAS  600 SIMULATION.
  var in EVENTSCH.PAS  601 SIMULATION.
  ref in EVENTSCH.PAS  603 SIMULATION.

system_data : system_description (VarParam)
```

```
  dec in EVENTSCH.PAS  259 SIMULATION.AREA_J\
  ref in EVENTSCH.PAS  262 SIMULATION.AREA_J\

system_data : system_description (VarParam)
  dec in EVENTSCH.PAS  329 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  338 SIMULATION.ARRIVAL\
  var in EVENTSCH.PAS  345 SIMULATION.ARRIVAL\

system_data : system_description (VarParam)
  dec in EVENTSCH.PAS  142 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  150 SIMULATION.INITIALIZE\

system_data : system_description (VarParam)
  dec in EVENTSCH.PAS  373 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  474 SIMULATION.PROCESS_EVENT\
  var in EVENTSCH.PAS  477 SIMULATION.PROCESS_EVENT\
  var in EVENTSCH.PAS  481 SIMULATION.PROCESS_EVENT\
  var in EVENTSCH.PAS  498 SIMULATION.PROCESS_EVENT\

system_data : system_description (VarParam)
  dec in EVENTSCH.PAS  445 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  449 SIMULATION.PROCESS_EVENT\DEPARTURE\
  var in EVENTSCH.PAS  453 SIMULATION.PROCESS_EVENT\DEPARTURE\

system_data : system_description (VarParam)
  dec in EVENTSCH.PAS  402 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  409 SIMULATION.PROCESS_EVENT\SCHEDULE\

SYSTEM_departure (Const)
  dec in EVENTSCH.PAS   83 SIMULATION.
  ref in EVENTSCH.PAS  476 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  491 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  494 SIMULATION.PROCESS_EVENT\

system_description = Record (Type)
  dec in EVENTSCH.PAS  105 SIMULATION.
  ref in EVENTSCH.PAS  122 SIMULATION.
  ref in EVENTSCH.PAS  142 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  259 SIMULATION.AREA_J\
  ref in EVENTSCH.PAS  329 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  373 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  402 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  445 SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref in EVENTSCH.PAS  550 SIMULATION.WRITE_RESULTS\

this_event : event_ptr (Var)
  dec in EVENTSCH.PAS  376 SIMULATION.PROCESS_EVENT\
  set in EVENTSCH.PAS  471 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  473 SIMULATION.PROCESS_EVENT\
  ref in EVENTSCH.PAS  476 SIMULATION.PROCESS_EVENT\
  var in EVENTSCH.PAS  477 SIMULATION.PROCESS_EVENT\
  var in EVENTSCH.PAS  495 SIMULATION.PROCESS_EVENT\
```

```
  var in EVENTSCH.PAS  498 SIMULATION.PROCESS_EVENT\

time = Real (Type)
  dec in EVENTSCH.PAS   79 SIMULATION.
  ref in EVENTSCH.PAS   88 SIMULATION.event_record.
  ref in EVENTSCH.PAS   97 SIMULATION.facility_record.
  ref in EVENTSCH.PAS  103 SIMULATION.
  ref in EVENTSCH.PAS  115 SIMULATION.system_description.
  ref in EVENTSCH.PAS  192 SIMULATION.RAND\
  ref in EVENTSCH.PAS  199 SIMULATION.RAND\
  ref in EVENTSCH.PAS  230 SIMULATION.EXPON\
  ref in EVENTSCH.PAS  230 SIMULATION.EXPON\
  ref in EVENTSCH.PAS  233 SIMULATION.EXPON\
  ref in EVENTSCH.PAS  333 SIMULATION.ARRIVAL\
  ref in EVENTSCH.PAS  405 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  559 SIMULATION.WRITE_RESULTS\

time_facility_available : time (Field)
  dec in EVENTSCH.PAS   95 SIMULATION.facility_record.
  set in EVENTSCH.PAS  162 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  412 SIMULATION.PROCESS_EVENT\SCHEDULE\
  set in EVENTSCH.PAS  413 SIMULATION.PROCESS_EVENT\SCHEDULE\
  set in EVENTSCH.PAS  415 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  415 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  421 SIMULATION.PROCESS_EVENT\SCHEDULE\

time_j_minus_1 : time (Field)
  dec in EVENTSCH.PAS  113 SIMULATION.system_description.
  set in EVENTSCH.PAS  157 SIMULATION.INITIALIZE\
  ref in EVENTSCH.PAS  263 SIMULATION.AREA_J\
  set in EVENTSCH.PAS  265 SIMULATION.AREA_J\
  set in EVENTSCH.PAS  461 SIMULATION.PROCESS_EVENT\DEPARTURE\

total_simulations (Const)
  dec in EVENTSCH.PAS   74 SIMULATION.
  ref in EVENTSCH.PAS  585 SIMULATION.

total_utilization_time : time (Field)
  dec in EVENTSCH.PAS   96 SIMULATION.facility_record.
  set in EVENTSCH.PAS  163 SIMULATION.INITIALIZE\
  set in EVENTSCH.PAS  418 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  418 SIMULATION.PROCESS_EVENT\SCHEDULE\
  ref in EVENTSCH.PAS  564 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  565 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  566 SIMULATION.WRITE_RESULTS\
  ref in EVENTSCH.PAS  567 SIMULATION.WRITE_RESULTS\

transient_length (Const)
  dec in EVENTSCH.PAS   75 SIMULATION.
  ref in EVENTSCH.PAS  458 SIMULATION.PROCESS_EVENT\DEPARTURE\

WRITE_HEADING (Proc)
```

```
  dec in EVENTSCH.PAS  517 SIMULATION.
  ref in EVENTSCH.PAS  584 SIMULATION.

WRITE_RESULTS (Proc)
  dec in EVENTSCH.PAS  550 SIMULATION.
  ref in EVENTSCH.PAS  603 SIMULATION.

z : time (Var)
  dec in EVENTSCH.PAS  199 SIMULATION.RAND\
  set in EVENTSCH.PAS  203 SIMULATION.RAND\
  set in EVENTSCH.PAS  204 SIMULATION.RAND\
  ref in EVENTSCH.PAS  204 SIMULATION.RAND\
  ref in EVENTSCH.PAS  205 SIMULATION.RAND\
  set in EVENTSCH.PAS  207 SIMULATION.RAND\
  ref in EVENTSCH.PAS  207 SIMULATION.RAND\
  ref in EVENTSCH.PAS  208 SIMULATION.RAND\
```

Appendix C. Identifier Report

The identifier report for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The report begins on the following page.

Identifier Report for
D:\RICK\THESIS\EVENTSCH.PAS
Saturday 9/10/88 at 13:03:51

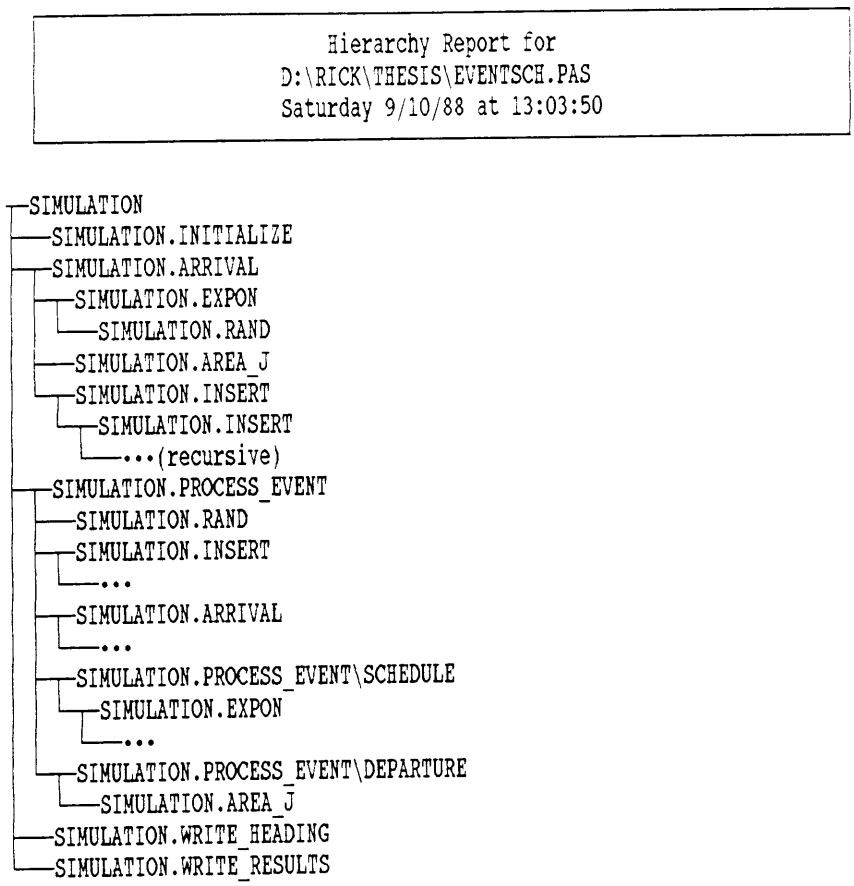
```
accumulated_time : time (Field)  dec in EVENTSCH.PAS 115 SIMULATION.system_description.
area : time (Field)  dec in EVENTSCH.PAS 114 SIMULATION.system_description.
AREA_J (Proc)  dec in EVENTSCH.PAS 259 SIMULATION.
ARRIVAL (Proc)  dec in EVENTSCH.PAS 328 SIMULATION.
clock : time (Field)  dec in EVENTSCH.PAS 111 SIMULATION.system_description.
completed_jobs : Integer (Field)  dec in EVENTSCH.PAS 108 SIMULATION.system_description.
CPU1_arrival (Const)  dec in EVENTSCH.PAS 82 SIMULATION.
CPU1_mean (Const)  dec in EVENTSCH.PAS 68 SIMULATION.
CPU1_probability (Const)  dec in EVENTSCH.PAS 71 SIMULATION.
CPU1_util : time (Var)  dec in EVENTSCH.PAS 554 SIMULATION.WRITE_RESULTS\
CPU2_arrival (Const)  dec in EVENTSCH.PAS 82 SIMULATION.
CPU2_mean (Const)  dec in EVENTSCH.PAS 69 SIMULATION.
CPU2_util : time (Var)  dec in EVENTSCH.PAS 555 SIMULATION.WRITE_RESULTS\
current_jobs : Integer (Field)  dec in EVENTSCH.PAS 107 SIMULATION.system_description.
d2p31m (Const)  dec in EVENTSCH.PAS 195 SIMULATION.RAND\
DEPARTURE (Proc)  dec in EVENTSCH.PAS 444 SIMULATION.PROCESS_EVENT\
device : input_device_type (Var)  dec in EVENTSCH.PAS 124 SIMULATION.
event : event_ptr (Var)  dec in EVENTSCH.PAS 332 SIMULATION.ARRIVAL\
event : event_ptr (VarParam)  dec in EVENTSCH.PAS 291 SIMULATION.INSERT\
event : event_ptr (VarParam)  dec in EVENTSCH.PAS 444 SIMULATION.PROCESS_EVENT\DEPARTURE\
event : event_ptr (VarParam)  dec in EVENTSCH.PAS 401 SIMULATION.PROCESS_EVENT\SCHEDULE\
event_counter : event_type (Var)  dec in EVENTSCH.PAS 146 SIMULATION.INITIALIZE\
event_list : event_ptr (Var)  dec in EVENTSCH.PAS 123 SIMULATION.
event_list : event_ptr (VarParam)  dec in EVENTSCH.PAS 328 SIMULATION.ARRIVAL\
event_list : event_ptr (VarParam)  dec in EVENTSCH.PAS 143 SIMULATION.INITIALIZE\
event_list : event_ptr (VarParam)  dec in EVENTSCH.PAS 291 SIMULATION.INSERT\
event_list : event_ptr (VarParam)  dec in EVENTSCH.PAS 372 SIMULATION.PROCESS_EVENT\
event_ptr = ^event_record (Type)  dec in EVENTSCH.PAS 85 SIMULATION.
event_record = Record (Type)  dec in EVENTSCH.PAS 85 SIMULATION.
event_type = Enumerated (Type)  dec in EVENTSCH.PAS 82 SIMULATION.
ex : time (Var)  dec in EVENTSCH.PAS 233 SIMULATION.EXPON\
EXPON : time (Function)  dec in EVENTSCH.PAS 230 SIMULATION.
exp_jobs : time (Var)  dec in EVENTSCH.PAS 557 SIMULATION.WRITE_RESULTS\
exp_time : time (Var)  dec in EVENTSCH.PAS 558 SIMULATION.WRITE_RESULTS\
facility : facility_type (Field)  dec in EVENTSCH.PAS 117 SIMULATION.system_description.
facility_record = Record (Type)  dec in EVENTSCH.PAS 94 SIMULATION.
facility_type = Array[Subrange] of facility_record (Type)  dec in EVENTSCH.PAS 100 SIMULATION.
INITIALIZE (Proc)  dec in EVENTSCH.PAS 142 SIMULATION.
input_devices = Array[Subrange] of time (Type)  dec in EVENTSCH.PAS 103 SIMULATION.
input_device_IAT : input_devices (Field)  dec in EVENTSCH.PAS 116 SIMULATION.system_description.
input_device_type = Enumerated (Type)  dec in EVENTSCH.PAS 80 SIMULATION.
INSERT (Proc)  dec in EVENTSCH.PAS 291 SIMULATION.
intquot : Integer (Var)  dec in EVENTSCH.PAS 200 SIMULATION.RAND\
in_steady_state : Boolean (Field)  dec in EVENTSCH.PAS 106 SIMULATION.system_description.
JES_arrival (Const)  dec in EVENTSCH.PAS 82 SIMULATION.
JES_mean (Const)  dec in EVENTSCH.PAS 67 SIMULATION.
```

```
JES_util : time (Var)  dec in EVENTSCH.PAS 553 SIMULATION.WRITE_RESULTS\
jobs_time_j_minus_1 : Integer (Field)  dec in EVENTSCH.PAS 109 SIMULATION.system_description.
job_origin : input_device_type (Field)  dec in EVENTSCH.PAS 90 SIMULATION.event_record.
LAN (Const)  dec in EVENTSCH.PAS 80 SIMULATION.
LAN_mean (Const)  dec in EVENTSCH.PAS 66 SIMULATION.
length_of_event : time (Var)  dec in EVENTSCH.PAS 405 SIMULATION.PROCESS_EVENT\SCHEDULE\
M1200 (Const)  dec in EVENTSCH.PAS 80 SIMULATION.
M1200_mean (Const)  dec in EVENTSCH.PAS 64 SIMULATION.
M2400 (Const)  dec in EVENTSCH.PAS 80 SIMULATION.
M2400_mean (Const)  dec in EVENTSCH.PAS 65 SIMULATION.
M300 (Const)  dec in EVENTSCH.PAS 80 SIMULATION.
M300_mean (Const)  dec in EVENTSCH.PAS 63 SIMULATION.
mean : time (Var)  dec in EVENTSCH.PAS 333 SIMULATION.ARRIVAL\
mean : time (ValParam)  dec in EVENTSCH.PAS 230 SIMULATION.EXPON\
mean_process_time : time (Field)  dec in EVENTSCH.PAS 97 SIMULATION.facility_record.
next_event : event_type (Field)  dec in EVENTSCH.PAS 89 SIMULATION.event_record.
next_event_time : time (Field)  dec in EVENTSCH.PAS 88 SIMULATION.event_record.
next_link : event_ptr (Field)  dec in EVENTSCH.PAS 91 SIMULATION.event_record.
origin : input_device_type (ValParam)  dec in EVENTSCH.PAS 328 SIMULATION.ARRIVAL\
output_items (Const)  dec in EVENTSCH.PAS 520 SIMULATION.WRITE_HEADING\
PRNT_arrival (Const)  dec in EVENTSCH.PAS 82 SIMULATION.
PRNT_mean (Const)  dec in EVENTSCH.PAS 70 SIMULATION.
PRNT_probability (Const)  dec in EVENTSCH.PAS 72 SIMULATION.
PRNT_util : time (Var)  dec in EVENTSCH.PAS 556 SIMULATION.WRITE_RESULTS\
PROCESS_EVENT (Proc)  dec in EVENTSCH.PAS 372 SIMULATION.
quotient : time (Var)  dec in EVENTSCH.PAS 198 SIMULATION.RAND\
r : time (Var)  dec in EVENTSCH.PAS 233 SIMULATION.EXPON\
RAND : time (Function)  dec in EVENTSCH.PAS 192 SIMULATION.
SCHEDULE (Proc)  dec in EVENTSCH.PAS 401 SIMULATION.PROCESS_EVENT\
seed : Integer (VarParam)  dec in EVENTSCH.PAS 230 SIMULATION.EXPON\
seed : Integer (VarParam)  dec in EVENTSCH.PAS 192 SIMULATION.RAND\
seed : Integer (Field)  dec in EVENTSCH.PAS 110 SIMULATION.system_description.
seed_const (Const)  dec in EVENTSCH.PAS 73 SIMULATION.
SIMULATION (Program)  dec in EVENTSCH.PAS 58 SIMULATION.
simulations : Integer (Var)  dec in EVENTSCH.PAS 121 SIMULATION.
statistics : system_description (ValParam)  dec in EVENTSCH.PAS 550 SIMULATION.WRITE_RESULTS\
steady_state_len (Const)  dec in EVENTSCH.PAS 76 SIMULATION.
steady_state_start : time (Field)  dec in EVENTSCH.PAS 112 SIMULATION.system_description.
steady_time : time (Var)  dec in EVENTSCH.PAS 559 SIMULATION.WRITE_RESULTS\
submit_time : time (Field)  dec in EVENTSCH.PAS 87 SIMULATION.event_record.
system_data : system_description (Var)  dec in EVENTSCH.PAS 122 SIMULATION.
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 259 SIMULATION.AREA_J\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 329 SIMULATION.ARRIVAL\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 142 SIMULATION.INITIALIZE\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 373 SIMULATION.PROCESS_EVENT\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 445 SIMULATION.PROCESS_EVENT\DEPARTURE\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 402 SIMULATION.PROCESS_EVENT\SCHEDULE\
SYSTEM_departure (Const)  dec in EVENTSCH.PAS 83 SIMULATION.
system_description = Record (Type)  dec in EVENTSCH.PAS 105 SIMULATION.
this_event : event_ptr (Var)  dec in EVENTSCH.PAS 376 SIMULATION.PROCESS_EVENT\
time = Real (Type)  dec in EVENTSCH.PAS 79 SIMULATION.
time_facility_available : time (Field)  dec in EVENTSCH.PAS 95 SIMULATION.facility_record.
```

```
time_j_minus_1 : time (Field)  dec in EVENTSCH.PAS 113 SIMULATION.system_description.
total_simulations (Const)  dec in EVENTSCH.PAS 74 SIMULATION.
total_utilization_time : time (Field)  dec in EVENTSCH.PAS 96 SIMULATION.facility_record.
transient_length (Const)  dec in EVENTSCH.PAS 75 SIMULATION.
WRITE_HEADING (Proc)  dec in EVENTSCH.PAS 517 SIMULATION.
WRITE_RESULTS (Proc)  dec in EVENTSCH.PAS 550 SIMULATION.
z : time (Var)  dec in EVENTSCH.PAS 199 SIMULATION.RAND\
```

Appendix D. Hierarchy Report

The hierarchy report for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The report is on the following page.


Hierarchy Report for
D:\RICK\THESIS\EVENTSCH.PAS
Saturday 9/10/88 at 13:03:50
SIMULATION
SIMULATION.INITIALIZE
SIMULATION.ARRIVAL
SIMULATION.EXPON
SIMULATION.RAND
SIMULATION.AREA_J
SIMULATION.INSERT
SIMULATION.INSERT
...(recursive)
SIMULATION.PROCESS_EVENT
SIMULATION.RAND
SIMULATION.INSERT
...
SIMULATION.ARRIVAL
...
SIMULATION.PROCESS_EVENT\SCHEDULE
SIMULATION.EXPON
...
SIMULATION.PROCESS_EVENT\DEPARTURE
SIMULATION.AREA_J
SIMULATION.WRITE_HEADING
SIMULATION.WRITE_RESULTS

Appendix E. Warning Report

The warning report for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The report is on the following page.

```
                Warning Report for
          D:\RICK\THESIS\EVENTSCH.PAS
          Saturday 9/10/88 at 13:03:50
```

```
Units Not Found

  MON

Identifiers Never Used

  r : time (Var)  dec in EVENTSCH.PAS 233 SIMULATION.EXPON\
  ex : time (Var)  dec in EVENTSCH.PAS 233 SIMULATION.EXPON\
```

Appendix F. Duplicate Identifier Report

The duplicate identifier report for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The report is on the following page.

```
                Duplicate Identifier Report for
                  D:\RICK\THESIS\EVENTSCH.PAS
                  Saturday 9/10/88 at 13:03:52

event : event_ptr (Var)  dec in EVENTSCH.PAS 332 SIMULATION.ARRIVAL\
event : event_ptr (VarParam)  dec in EVENTSCH.PAS 291 SIMULATION.INSERT\
event : event_ptr (VarParam)  dec in EVENTSCH.PAS 444 SIMULATION.PROCESS_EVENT\DEPARTURE\
event : event_ptr (VarParam)  dec in EVENTSCH.PAS 401 SIMULATION.PROCESS_EVENT\SCHEDULE\

event_list : event_ptr (Var)  dec in EVENTSCH.PAS 123 SIMULATION.
event_list : event_ptr (VarParam)  dec in EVENTSCH.PAS 328 SIMULATION.ARRIVAL\
event_list : event_ptr (VarParam)  dec in EVENTSCH.PAS 143 SIMULATION.INITIALIZE\
event_list : event_ptr (VarParam)  dec in EVENTSCH.PAS 291 SIMULATION.INSERT\
event_list : event_ptr (VarParam)  dec in EVENTSCH.PAS 372 SIMULATION.PROCESS_EVENT\

mean : time (Var)  dec in EVENTSCH.PAS 333 SIMULATION.ARRIVAL\
mean : time (ValParam)  dec in EVENTSCH.PAS 230 SIMULATION.EXPON\

seed : Integer (VarParam)  dec in EVENTSCH.PAS 230 SIMULATION.EXPON\
seed : Integer (VarParam)  dec in EVENTSCH.PAS 192 SIMULATION.RAND\
seed : Integer (Field)  dec in EVENTSCH.PAS 110 SIMULATION.system_description.

system_data : system_description (Var)  dec in EVENTSCH.PAS 122 SIMULATION.
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 259 SIMULATION.AREA_J\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 329 SIMULATION.ARRIVAL\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 142 SIMULATION.INITIALIZE\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 373 SIMULATION.PROCESS_EVENT\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 445 SIMULATION.PROCESS_EVENT\DEPARTURE\
system_data : system_description (VarParam)  dec in EVENTSCH.PAS 402 SIMULATION.PROCESS_EVENT\SCHEDULE\
```

Appendix G. Side Effects Report

The side effects report for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The report is on the following page.

```
                    Side Effects Report for
                  D:\RICK\THESIS\EVENTSCH.PAS
                  Saturday 9/10/88 at 13:03:50
```

```
SIMULATION.INITIALIZE\
  ref SIMULATION.M300_mean
  ref SIMULATION.M1200_mean
  ref SIMULATION.M2400_mean
  ref SIMULATION.LAN_mean
  ref SIMULATION.JES_mean
  ref SIMULATION.CPU1_mean
  ref SIMULATION.CPU2_mean
  ref SIMULATION.PRNT_mean
  ref SIMULATION.M300
  ref SIMULATION.M1200
  ref SIMULATION.M2400
  ref SIMULATION.LAN
  ref SIMULATION.JES_arrival
  ref SIMULATION.CPU1_arrival
  ref SIMULATION.CPU2_arrival
  ref SIMULATION.PRNT_arrival

SIMULATION.ARRIVAL\
  ref SIMULATION.JES_arrival

SIMULATION.PROCESS_EVENT\
  ref SIMULATION.CPU1_probability
  ref SIMULATION.PRNT_probability
  ref SIMULATION.JES_arrival
  ref SIMULATION.CPU1_arrival
  ref SIMULATION.CPU2_arrival
  ref SIMULATION.PRNT_arrival
  ref SIMULATION.SYSTEM_departure

SIMULATION.PROCESS_EVENT\DEPARTURE\
  ref SIMULATION.transient_length

SIMULATION.WRITE_RESULTS\
  ref SIMULATION.steady_state_len
  ref SIMULATION.JES_arrival
  ref SIMULATION.CPU1_arrival
  ref SIMULATION.CPU2_arrival
  ref SIMULATION.PRNT_arrival
```

Appendix H. Totals Report

The totals report for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The report is on the following page.

Totals Report for
D:\RICK\THESIS\EVENTSCH.PAS
Saturday 9/10/88 at 13:03:50

Class	Total	Global	Interfaced	Unused
Units	1	--	--	--
Files	1	--	--	--
Lines	605	--	--	--
Procedures	9	7	0	0
Functions	2	2	0	0
Constants	25	23	0	0
Types	9	9	0	0
Labels	0	0	0	--
Variables	21	4	0	2
Parameters	3	--	--	0
Var params	15	--	--	0

Unit	Accepted	Found
MON	Yes	No

Appendix I. Sorted Procedure Index Report

The sorted procedure index report for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The report is on the following page.

Sorted Procedure Index Report for
D:\RICK\THESIS\EVENTSCH.PAS
Saturday 9/10/88 at 13:03:51

Legend: P=Program U=Unit I=Interfaced *=Not Called !=Exposed

Appendix J. SIMULATION Execution Profile

The execution profile for the programmed model SIMULATION was created using Turbo Analyst 4.0™ from TurboPower Software of Scotts Valley, CA. The report begins on the following page.

```
                    * Hit Count for All Statements *

Statistics
================================================================================
Total statements:        158  Statements in range:        158  100.00%
Total hits:          7018617  Hits in range:          7018617  100.00%
Highest:              703525  High in range:           703525  100.00%
Not executed:              1  Number in range:              1  100.00%

Hit Count     D:\RICK\THESIS\EVENTSCH.PAS
================================================================================
              (*****************************************************************************)
              (*                                                                           *)
              (*  P R O G R A M    S I M U L A T I O N                                     *)
              (*                                                                           *)
              (*  DESCRIPTION:  Program SIMULATION simulates a computer system that        *)
              (*                operates in batch mode, with jobs arriving from four       *)
              (*                different types of entry devices to a job entry            *)
              (*                scheduler (JES).  Upon processing by the JES, a job        *)
              (*                will be scheduled to execute on one of two CPUs, and       *)
              (*                upon exit from the CPU, will either proceed to a           *)
              (*                printer device (PRNT) or will leave the simulated          *)
              (*                system to a virtual reader.  Jobs leaving the printer      *)
              (*                go directly to the virtual reader.                         *)
              (*                                                                           *)
              (*                This program 'tracks' the flow of jobs through the         *)
              (*                system using the event scheduling  world view.             *)
              (*                Utilization of the various resources is measured, as       *)
              (*                is the total waiting time of jobs in the system (for       *)
              (*                computing expected waiting time).                          *)
              (*                                                                           *)
              (*                The simulation is repeated a specified number of           *)
              (*                times, with each simulation consisting of a transient      *)
              (*                state and a steady state.  Measurements are not taken      *)
              (*                until the last transient state job leaves the system,      *)
              (*                at which time all devices are checked for any current      *)
              (*                use, utilization (if any) recorded, and measurement        *)
              (*                continues until the last steady state job leaves the       *)
              (*                system.  All measurement ceases at that point in           *)
              (*                time.                                                      *)
              (*                                                                           *)
              (*  HISTORY                                                                  *)
              (*    Created By     : Richard B. Whitner                                    *)
              (*    Date Created   : 05/05/87                                              *)
              (*    Revised By     : Richard B. Whitner                                    *)
              (*    Date Revised   : 12/21/87                                              *)
              (*    Revision Notes : Program modified to improve runtime efficiency        *)
              (*                     and to simplify the model representation.             *)
              (*                                                                           *)
              (*  INPUTS:   none.                                                          *)
              (*
```

```
(*  OUTPUTS:  Output (directed to standard output) consists of perform-   *)
(*            ance statistics formatted in a manner acceptable to         *)
(*            CIMULT, a program which will construct confidence           *)
(*            intervals from the generated performance measures.  The     *)
(*            performance measures generated consist of utilization       *)
(*            rate of each system facility, expected system waiting       *)
(*            time, and expected jobs in the system.                      *)
(*                                                                        *)
(*  CALLS:    procedure INITIALIZE                                        *)
(*            procedure WRITE_HEADING                                     *)
(*            procedure WRITE_RESULTS                                     *)
(*            procedure ARRIVAL                                           *)
(*            procedure PROCESS_EVENTS                                    *)
(*                                                                        *)
(**************************************************************************)
{$D+}
{$T+}
program SIMULATION (output);

  uses MON;                              (* allow instrumentation          *)

  const
     M300_mean         = 3200.0;     (* mean dist IAT for Modem 300 user  *)
     M1200_mean        = 640.0;      (* mean dist IAT for Modem 1200 user *)
     M2400_mean        = 1600.0;     (* mean dist IAT for Modem 2400 user *)
     LAN_mean          = 266.67;     (* mean dist IAT for LAN user        *)
     JES_mean          = 112.0;      (* mean dist process time for JES    *)
     CPU1_mean         = 226.67;     (* mean dist process time for CPU1   *)
     CPU2_mean         = 300.0;      (* mean dist process time for CPU2   *)
     PRNT_mean         = 160.0;      (* mean dist process time for PRNT   *)
     CPU1_probability  = 0.6;        (* probability of job using CPU1     *)
     PRNT_probability  = 0.8;        (* probability of job using printer  *)
     seed_const        = 15987;      (* used to generate a random number  *)
     total_simulations = 1;          (* number of simulation repetitions  *)
     transient_length  = 3000;       (* number of jobs in transient state *)
     steady_state_len  = 15000;      (* number of jobs in steady state    *)

  type
     time             = real;                (* simulation time typedef *)
     input_device_type = (M300,M1200,M2400,LAN);  (* input device types   *)

     event_type = (JES_arrival,CPU1_arrival,CPU2_arrival,PRNT_arrival,
                   SYSTEM_departure); (* enumerated event types          *)

     event_ptr    = ^event_record;    (* pointer to the event list record *)
     event_record = record
        submit_time,                  (* job submit time                 *)
        next_event_time : time;       (* sched time to execute next event *)
        next_event      : event_type; (* next event to perform           *)
        job_origin      : input_device_type; (* device job came from     *)
        next_link       : event_ptr   (* pointer to next event in list   *)
     end;
```

```
    facility_record = record          (* data record for each facility    *)
       time_facility_available,       (* next time facility is idle       *)
       total_utilization_time,        (* accumulated utilization time     *)
       mean_process_time      : time (* mean distributed process time    *)
    end;

    facility_type = array[JES_arrival..PRNT_arrival] of facility_record;
    (* events with an associated facility in this simulation *)

    input_devices = array[M300..LAN] of time;    (* enumerated array     *)

    system_description = record       (* structure to hold system context *)
       in_steady_state    : boolean; (* steady state start flag          *)
       current_jobs,                 (* number of jobs in system now     *)
       completed_jobs,               (* number jobs having left system   *)
       jobs_time_j_minus_1,          (* #jobs, as of last arrival/depart *)
       seed               : integer; (* random number generator seed     *)
       clock,                        (* system simulation clock          *)
       steady_state_start,           (* time the steady state began      *)
       time_j_minus_1,               (* time of the last arrival/depart  *)
       area,                         (* jobs-in-sys / time unit curve    *)
       accumulated_time   : time;    (* total job waiting time           *)
       input_device_IAT   : input_devices; (* an IAT for each job source *)
       facility           : facility_type  (* device information        *)
    end;

  var
    simulations : integer;            (* count of number of simulations  *)
    system_data : system_description; (* all pertinent system data       *)
    event_list  : event_ptr;          (* current system event list       *)
    device      : input_device_type;  (* enumerated counter for loops    *)


(*************************************************************************)
(*                                                                       *)
(*  P R O C E D U R E    I N I T I A L I Z E                             *)
(*                                                                       *)
(*  DESCRIPTION:  Procedure INITIALIZE sets all system variables to their *)
(*                appropriate starting values.                           *)
(*                                                                       *)
(*  Procedure Calls : none.                                              *)
(*  Function Calls  : none.                                              *)
(*  Called By       : program SIMULATION.                                *)
(*  Parameters      : system_data -- contains pertinent system data       *)
(*                    event_list  -- system event list                   *)
(*                                                                       *)
(*************************************************************************)

  procedure INITIALIZE(var system_data : system_description;
                       var event_list  : event_ptr);
```

```
                var
                  event_counter : event_type;          (* enumerated loop counter *)

[       1]      begin
[       1]        event_list := nil;
[       1]        with system_data do begin
[       1]           in_steady_state     := false;
[       1]           current_jobs        := 0;
[       1]           completed_jobs      := 0;
[       1]           jobs_time_j_minus_1 := 0;
[       1]           clock               := 0.0;
[       1]           steady_state_start  := 0.0;
[       1]           time_j_minus_1      := 0.0;
[       1]           area                := 0.0;
[       1]           accumulated_time    := 0.0;
[       1]           for event_counter := JES_arrival to PRNT_arrival do
[       4]              with facility[event_counter] do begin
[       4]                 time_facility_available := 0.0;
[       4]                 total_utilization_time  := 0.0;
[       4]              end;
[       1]           facility[ JES_arrival].mean_process_time := JES_mean;
[       1]           facility[CPU1_arrival].mean_process_time := CPU1_mean;
[       1]           facility[CPU2_arrival].mean_process_time := CPU2_mean;
[       1]           facility[PRNT_arrival].mean_process_time := PRNT_mean;
[       1]           input_device_IAT[ M300]                  := M300_mean;
[       1]           input_device_IAT[M1200]                  := M1200_mean;
[       1]           input_device_IAT[M2400]                  := M2400_mean;
[       1]           input_device_IAT[  LAN]                  := LAN_mean;
                   end;
[       1]      end;  (* procedure INITIALIZE *)


                (*************************************************************************)
                (*                                                                       *)
                (*  F U N C T I O N    R A N D                                           *)
                (*                                                                       *)
                (*  DESCRIPTION:  Function RAND returns a pseudo-random number N between *)
                (*                0 (inclusive) and 1, based on the seed parameter.      *)
                (*                                                                       *)
                (*  Procedure Calls : none.                                              *)
                (*  Function Calls  : none.                                              *)
                (*  Called By       : function EXPON.                                    *)
                (*                    procedure PROCESS_EVENT.                           *)
                (*  Parameters      : seed -- random number generator seed value.        *)
                (*                                                                       *)
                (*************************************************************************)

                  function RAND(var seed : integer) : time;

                    const
                      d2p31m = 2147483647.0;
```

```
                var
                   quotient,
                   z         : time;
                   intquot   : integer;

[    104375]     begin
[    104375]        z        := seed;
[    104375]        z        := 16807.0 * z;
[    104375]        quotient := z / d2p31m;
[    104375]        intquot  := trunc(quotient);
[    104375]        z        := z - intquot * d2p31m;
[    104375]        seed     := trunc(z);
                  { RAND     := z /d2p31m  }
[    104375]        RAND     := RANDOM
[    104375]     end;  (* function RAND *)


               (*****************************************************************************)
               (*                                                                           *)
               (*  F U N C T I O N    E X P O N                                             *)
               (*                                                                           *)
               (*  DESCRIPTION:  Function EXPON generates exponentially distributed         *)
               (*                values based on a given mean.                              *)
               (*                                                                           *)
               (*  Procedure Calls : none.                                                  *)
               (*  Function Calls  : RAND.                                                  *)
               (*  Called By       : procedure ARRIVAL.                                     *)
               (*                    procedure SCHEDULE.                                    *)
               (*  Parameters      : mean -- mean distribution value                        *)
               (*                    seed -- random number generator seed value             *)
               (*                                                                           *)
               (*****************************************************************************)

                function EXPON(mean : time;var seed : integer) : time;

                  var
                     r, ex : time;

[     68343]     begin
[     68343]        EXPON := -mean * LN(RAND(seed))
[     68343]     end;  (* function EXPON *)


               (*****************************************************************************)
               (*                                                                           *)
               (*  P R O C E D U R E    A R E A _ J                                         *)
               (*                                                                           *)
               (*  DESCRIPTION:  Procedure AREA_J computes the area beneath the curve       *)
               (*                given by the number of jobs in the system at time t.       *)
               (*                In this particular instance, the area computed is that     *)
               (*                of all jobs in the system since the last computation       *)
               (*                of the area (i.e., whenever there is an arrival to or      *)
```

```
                (*              departure from the system).                     *)
                (*                                                              *)
                (*  Procedure Calls : none.                                     *)
                (*  Function Calls  : none.                                     *)
                (*  Called By       : procedure ARRIVAL.                        *)
                (*                    procedure DEPARTURE.                      *)
                (*  Parameters      : system_data -- pertinent system data      *)
                (*                                                              *)
                (******************************************************************)

                 procedure AREA_J(var system_data : system_description);

[     29995]       begin
[     29995]          with system_data do begin
[     29995]             area := area + jobs_time_j_minus_1 * (clock - time_j_minus_1);
[     29995]             jobs_time_j_minus_1 := current_jobs;
[     29995]             time_j_minus_1      := clock
                    end
[     29995]       end;  (* procedure AREA_J *)


                (******************************************************************)
                (*                                                                *)
                (*  P R O C E D U R E    I N S E R T                              *)
                (*                                                                *)
                (*  DESCRIPTION:  Procedure INSERT places an event in the proper position *)
                (*                in the system event list, in order of ascending time.  *)
                (*                INSERT compares the time of the item to be inserted    *)
                (*                against the next item in the event list and calls      *)
                (*                itself recursively as needed until the proper location *)
                (*                is found.                                              *)
                (*                                                                *)
                (*  Procedure Calls : INSERT (recursively)                        *)
                (*  Function Calls  : none.                                       *)
                (*  Called By       : procedure INSERT (recursively)              *)
                (*                    procedure ARRIVAL.                          *)
                (*                    procedure PROCESS_EVENT.                    *)
                (*  Parameters      : event      -- the event to be inserted      *)
                (*                    event_list -- the current events list       *)
                (*                                                                *)
                (******************************************************************)

                 procedure INSERT(var event,event_list : event_ptr);

[    703525]       begin
[    703525]          if event_list = nil then begin    (* list empty, insert at the top *)
[      4512]             event^.next_link := event_list;
[      4512]             event_list       := event
[      4512]          end
                       else
[    699013]             if event^.next_event_time >= event_list^.next_event_time then
[    635182]                INSERT(event,event_list^.next_link)
```

```
                    else begin
[     63831]              event^.next_link := event_list;
[     63831]              event_list       := event
                    end
[    703525]       end;  (* procedure INSERT *)


              (*************************************************************************)
              (*                                                                       *)
              (*  P R O C E D U R E    A R R I V A L                                   *)
              (*                                                                       *)
              (*  DESCRIPTION:  Procedure ARRIVAL generates an arrival event from the  *)
              (*                same type of device that generated the job now calling *)
              (*                ARRIVAL.  The submit time is recorded, other necessary *)
              (*                values set, and the new event inserted into the event  *)
              (*                list.                                                  *)
              (*                                                                       *)
              (*  Procedure Calls : INSERT                                             *)
              (*  Function Calls  : none.                                              *)
              (*  Called By       : program SIMULATION.                                *)
              (*                    procedure PROCESS_EVENT.                           *)
              (*  Parameters      : origin      -- type of device generating an arrival *)
              (*                    event_list  -- the current events list             *)
              (*                    system_data -- pertinent system data               *)
              (*                                                                       *)
              (*************************************************************************)

                procedure ARRIVAL(origin : input_device_type;var event_list : event_ptr;
                                  var system_data : system_description);

                  var
                     event : event_ptr;        (* a new event to be added to the system *)
                     mean  : time;             (* mean distributed IAT time for device  *)

[     18020]       begin
[     18020]          NEW(event);
[     18020]          with event^ do
[     18020]             with system_data do begin
[     18020]                job_origin      := origin;
[     18020]                mean            := input_device_IAT[origin];
[     18020]                submit_time     := EXPON(mean,seed) + clock;
[     18020]                next_event_time := submit_time;
[     18020]                next_event      := JES_arrival;
[     18020]                if in_steady_state then
[     14995]                   AREA_J(system_data)
                          end;
[     18020]           INSERT(event,event_list)
[     18020]       end;  (* procedure ARRIVAL *)


              (*************************************************************************)
              (*                                                                       *)
```

```
(*  P R O C E D U R E    P R O C E S S _ E V E N T                    *)
(*                                                                    *)
(*  DESCRIPTION:  Procedure PROCESS_EVENT determines which type of event  *)
(*                occurs next for a given job and then SCHEDULEs that     *)
(*                event and INSERTs the job back into the current events  *)
(*                list, provided the next event is not a system departure *)
(*                event.                                                  *)
(*                                                                        *)
(*  Procedure Calls : ARRIVAL.                                            *)
(*                    DEPARTURE.                                          *)
(*                    INSERT.                                             *)
(*                    SCHEDULE.                                           *)
(*  Function Calls  : RAND.                                               *)
(*  Called By       : program SIMULATION.                                 *)
(*  Parameters      : event_list  -- the current events list              *)
(*                    system_data -- pertinent system data                *)
(*                                                                        *)
(**************************************************************************)

  procedure PROCESS_EVENT(var event_list  : event_ptr;
                          var system_data : system_description);

    var
      this_event : event_ptr;                    (* the current event *)


(**************************************************************************)
(*                                                                        *)
(*  P R O C E D U R E    S C H E D U L E                                  *)
(*                                                                        *)
(*  DESCRIPTION:  Procedure SCHEDULE is a generic scheduling routine      *)
(*                which performs the scheduling tasks common to all       *)
(*                events.  It determines if the event scheduled to occur  *)
(*                can take place at the scheduled time, and if it can't,  *)
(*                it determines when it could occur.  In either case,     *)
(*                the length of the event is determined, and then the     *)
(*                next event is scheduled, all of this relative to when   *)
(*                the event actually occurred.                            *)
(*                                                                        *)
(*  Procedure Calls : none.                                               *)
(*  Function Calls  : EXPON.                                              *)
(*  Called By       : program SIMULATION.                                 *)
(*                    procedure PROCESS_EVENT.                            *)
(*  Parameters      : event       -- the event to be scheduled            *)
(*                    system_data -- pertinent system data                *)
(*                                                                        *)
(**************************************************************************)

    procedure SCHEDULE(var event : event_ptr;
                       var system_data : system_description);

    var
```

```
                length_of_event : time;    (* the amount of time to complete event *)

[     50323]    begin
[     50323]      with event^ do
[     50323]        with system_data do
[     50323]          with facility[next_event] do begin
[     50323]            length_of_event := EXPON(mean_process_time,seed);
[     50323]            if time_facility_available <= next_event_time then
[     11277]              time_facility_available := next_event_time + length_of_event
                    else
[     39046]              time_facility_available := time_facility_available +
                                                     length_of_event;
[     50323]            if in_steady_state then
[     41884]              total_utilization_time := total_utilization_time +
                                                    length_of_event;
[     50323]            next_event_time := time_facility_available
                    end
[     50323]    end;  (* procedure SCHEDULE *)


          (*************************************************************************)
          (*                                                                       *)
          (*  P R O C E D U R E    D E P A R T U R E                                 *)
          (*                                                                       *)
          (*  DESCRIPTION:  Procedure DEPARTURE handles the exiting of a job from  *)
          (*                the system, checking to see whether or not the system  *)
          (*                has entered the steady state.  As soon as the system   *)
          (*                enters the steady state, all statistical values are    *)
          (*                initialized and the system flagged as being in the     *)
          (*                steady state.                                          *)
          (*                                                                       *)
          (*  Procedure Calls : AREA_J.                                            *)
          (*  Function Calls  : none.                                              *)
          (*  Called By       : procedure PROCESS_EVENT.                           *)
          (*  Parameters      : event       -- the departing event                 *)
          (*                    system_data -- pertinent system data               *)
          (*                                                                       *)
          (*************************************************************************)

              procedure DEPARTURE(var event : event_ptr;
                                  var system_data : system_description);


[     18000]    begin
[     18000]      with event^ do
[     18000]        with system_data do begin
[     18000]          current_jobs   := current_jobs - 1;
[     18000]          completed_jobs := completed_jobs + 1;
[     18000]          if in_steady_state then begin
[     15000]            AREA_J(system_data);
[     15000]            accumulated_time := accumulated_time + (next_event_time -
                                              submit_time)
[     15000]          end
```

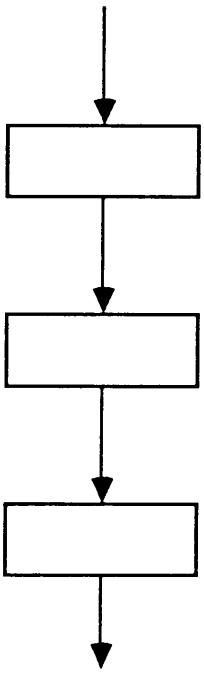
```
                    else
[      3000]                if completed_jobs = transient_length then begin
[         1]                   in_steady_state     := true;
[         1]                   steady_state_start  := clock;
[         1]                   time_j_minus_1      := clock;
[         1]                   jobs_time_j_minus_1 := current_jobs;
[         1]                   completed_jobs      := 0
                            end
                       end;
[     18000]           DISPOSE(event)
[     18000]        end;  (* procedure DEPARTURE *)


[     68323]     begin (* procedure PROCESS_EVENT *)
[     68323]        this_event := event_list;
[     68323]        event_list := event_list^.next_link;
[     68323]        with this_event^ do
[     68323]           with system_data do begin
[     68323]              clock := next_event_time;              (* advance system clock *)
[     68323]              if this_event^.next_event <> SYSTEM_departure then begin
[     50323]                 SCHEDULE(this_event,system_data);
[     50323]                 case next_event of
[     50323]                    JES_arrival : begin
[     18016]                       current_jobs := current_jobs + 1;
[     18016]                       ARRIVAL(job_origin,event_list,system_data);
[     18016]                       if RAND(seed) < CPU1_probability then
[     10785]                          next_event := CPU1_arrival
                                  else
[      7231]                          next_event := CPU2_arrival
[     18016]                    end;
[     32307]                    CPU1_arrival,CPU2_arrival :
[     18016]                       if RAND(seed) < PRNT_probability then
[     14300]                          next_event := PRNT_arrival
                                  else
[      3716]                          next_event := SYSTEM_departure;
[     14291]                    PRNT_arrival :
[     14291]                       next_event := SYSTEM_departure
                              end;
[     50323]                    INSERT(this_event,event_list)
[     50323]              end
                            else
[     18000]                    DEPARTURE(this_event,system_data);
                         end
[     68323]     end;  (* procedure PROCESS_EVENT *)


                (*************************************************************************)
                (*                                                                       *)
                (*  P R O C E D U R E    W R I T E _ H E A D I N G                       *)
                (*                                                                       *)
                (*  DESCRIPTION:  Procedure WRITE_HEADING writes the output heading in   *)
                (*                a form which can be utilized by CIMULT.                *)
```

```
(*                                                                          *)
(*  Procedure Calls : none.                                                 *)
(*  Function Calls  : none.                                                 *)
(*  Called By       : program SIMULATION.                                   *)
(*  Parameters      : none.                                                 *)
(*                                                                          *)
(****************************************************************************)

  procedure WRITE_HEADING;

    const
      output_items = 6;

[       1]   begin
[       1]     writeln(output_items:2);
[       1]     writeln('Utilization of the Job Entry Subsystem Scheduler');
[       1]     writeln('Utilization of CPU 1');
[       1]     writeln('Utilization of CPU 2');
[       1]     writeln('Utilization of the Printer');
[       1]     writeln('Expected Waiting Time');
[       1]     writeln('Expected Jobs in the System')
[       1]   end;  (* procedure WRITE_HEADING *)


(****************************************************************************)
(*                                                                          *)
(*  P R O C E D U R E    W R I T E _ R E S U L T S                          *)
(*                                                                          *)
(*  DESCRIPTION:  Procedure WRITE_RESULTS writes the results of the         *)
(*                simulation in a form which can be utilized by CIMULT.     *)
(*                Using the accumulated system statistics, all appro-       *)
(*                priate values are calculated before any output is         *)
(*                generated.                                                *)
(*                                                                          *)
(*  Procedure Calls : none.                                                 *)
(*  Function Calls  : none.                                                 *)
(*  Called By       : program SIMULATION.                                   *)
(*  Parameters      : statistics -- system statistical totals               *)
(*                                                                          *)
(****************************************************************************)

procedure WRITE_RESULTS(statistics : system_description);

  var
    JES_util,       (* Job Entry Scheduler utilization time                *)
    CPU1_util,      (* CPU1 utilization time                               *)
    CPU2_util,      (* CPU2 utilization time                               *)
    PRNT_util,      (* Printer utilization time                            *)
    exp_jobs,       (* Expected (avg.) number of jobs at any given time    *)
    exp_time,       (* Expected (avg.) waiting time for any given job      *)
    steady_time : time;           (* system time in the steady state       *)
```

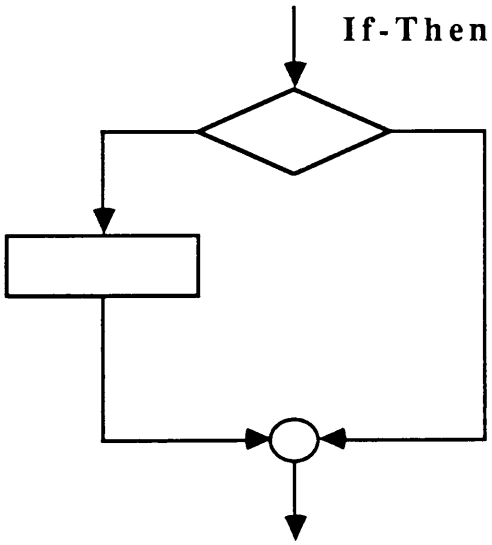
```
[        1]  begin
               with statistics do begin
[        1]       steady_time := clock - steady_state_start;
[        1]       JES_util  := facility[ JES_arrival].total_utilization_time / steady_time;
[        1]       CPU1_util := facility[CPU1_arrival].total_utilization_time / steady_time;
[        1]       CPU2_util := facility[CPU2_arrival].total_utilization_time / steady_time;
[        1]       PRNT_util := facility[PRNT_arrival].total_utilization_time / steady_time;
[        1]       exp_time  := area / steady_state_len;
[        1]       exp_jobs  := area / steady_time
               end;
[        1]    write(JES_util:8:5);
[        1]    write(CPU1_util:13:5);
[        1]    write(CPU2_util:13:5);
[        1]    write(PRNT_util:13:5);
[        1]    write(exp_time:17:5);
[        1]    writeln(exp_jobs:15:5)
[        1]  end;  (* procedure WRITE_RESULTS *)


               (*****************************************************************************)
               (*            M A I N     P R O G R A M     D R I V E R                      *)
               (*****************************************************************************)
[        0]* BEGIN  (* program SIMULATION *)
[        1]    WRITE_HEADING;
[        1]    for simulations := 1 to total_simulations do begin
                 with system_data do
[        1]         seed := simulations * seed_const + (simulations mod 2) + 1;
[        1]       INITIALIZE(system_data,event_list);

                 (************************************************)
                 (* Schedule the first job from each entry device *)
                 (************************************************)
[        1]       for device := M300 to LAN do
[        4]         ARRIVAL(device,event_list,system_data);

                 (************************************************)
                 (* Process the specified number of transient and *)
                 (* steady-state jobs.                            *)
                 (************************************************)
[    68324]       while system_data.completed_jobs < steady_state_len do
[    68323]         PROCESS_EVENT(event_list,system_data);

[        1]       WRITE_RESULTS(system_data)
[        1]    end
[        1]  END.  (* program SIMULATION *)
```

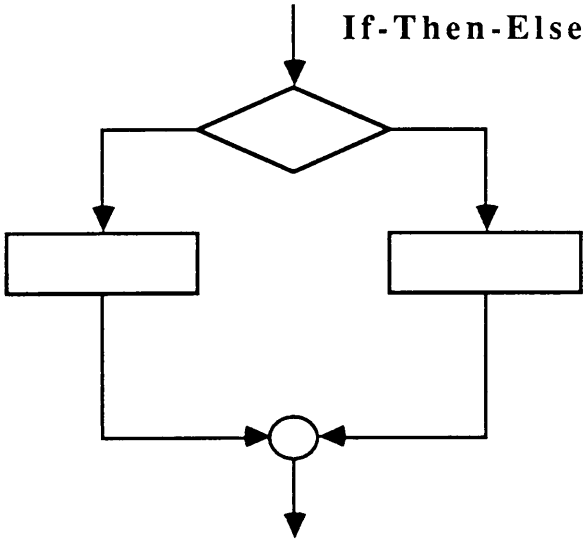
Appendix K. Control Structure Flowcharts

1. Sequence

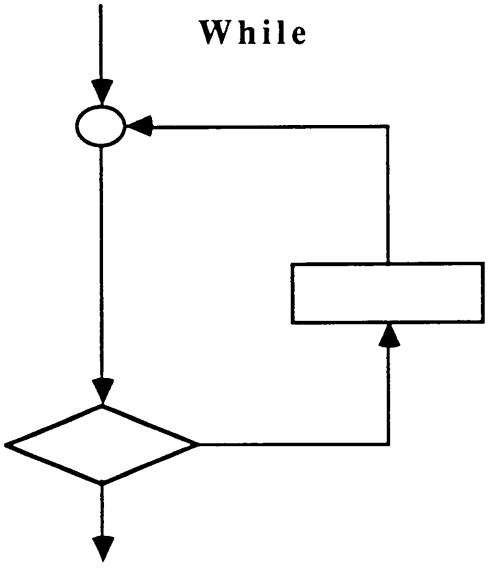


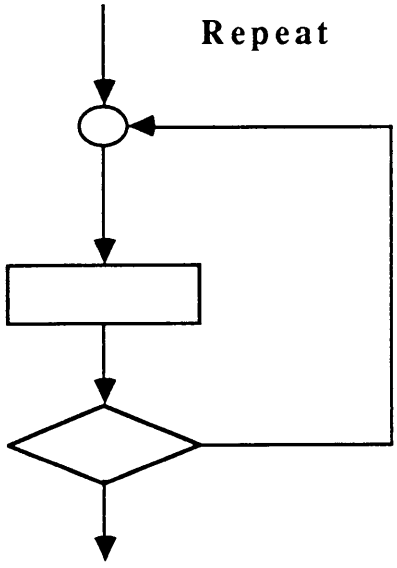
2. Selection





3. Iteration





The vita has been removed from the scanned document