Technical Report TR-89-17

GUIDELINES FOR SELECTING AND USING SIMULATION MODEL VERIFICATION TECHNIQUES

by

Richard B. Whitner and Osman Balci

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

15 August 1989

† An abridged version of this technical report will appear in *Proceedings of the 1989 Winter Simulation Conference,* Washington, D.C., December 4-6, 1989.

ABSTRACT

There is a lack of sufficient understanding and realization of the importance of simulation model verification in the simulation community. The demands placed on the software which serves as a computer-executable representation of a simulation model are increasing. In the field of software engineering, there is an abundance of software verification techniques that are applicable for simulation model verification. This paper is intended to reduce the communication gap between the software engineering and simulation communities by presenting software verification techniques applicable for simulation model verification in a terminology understandable by a simulationist. A taxonomy of verification techniques is developed to guide the simulationist in selecting and using such techniques. Characteristics, advantages, and disadvantages of verification techniques under each category are described.

CR Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification; D.2.5. [**Software Engineering**]: Testing and Debugging; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

Additional Key Words and Phrases: Credibility assessment, model quality assurance, simulation model verification, software verification

1. 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.

In the area of simulation, verification is also a crucial process. The simulation model life cycle, monitored by 13 credibility assessment stages (CASs), has a much broader scope than does the general software life cycle [Balci 1989]. This paper deals with the Programmed Model Verification (PMV) which is one of the 13 CASs.

There is much confusion concerning the difference between validation and verification. *Validation* is substantiating that the input-output transformation of the model has sufficient accuracy in representing the input-output transformation of the system. Whenever a model or model component is *compared* with reality, validation is performed. *Verification*, on the other hand, is substantiating that a simulation model is translated from one form into another, during its development life cycle, with sufficient accuracy. PMV is substantiating that the Programmed Model (the executable simulation model) is translated from the simulation model specification (Communicative Model) with sufficient accuracy. Validation deals with building the right model, verification deals with building the model right [Boehm 1984].

The perfunctory view of the PMV has caused many simulation experts to overlook the area of PMV. Quite often, neither sufficient time nor resources are allocated for it. As one simulation study from the aerospace industry admitted, PMV 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 a simulation study for extended periods of time.

All software verification techniques are applicable to PMV. 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 paper is intended to reduce the communication gap between the software engineering and simulation communities by presenting software verification techniques applicable for PMV in a terminology understandable by a simulationist.

Section 2 presents a taxonomy for PMV techniques. Software verification techniques are described for PMV in Section 3. Concluding remarks are given in Section 4.

2. A TAXONOMY FOR SIMULATION MODEL VERIFICATION TECHNIQUES

Programmed Model (executable simulation model) Verification 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 paper categorizes the verification process into six distinct verification perspectives: informal, static, dynamic, symbolic, constraint, and formal analysis. The taxonomy is shown in Figure 1. 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, 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.

Table 1 summarizes a number of characteristics of the general nature of each category: (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) whether or not the category is considered instrumentation-based, and (8) the relative importance of the associated techniques to PMV. The comparison among the categories (e.g., Level of Formality) is intended more to give a relative view among the spectrum of categories rather than to measure against some known standard.

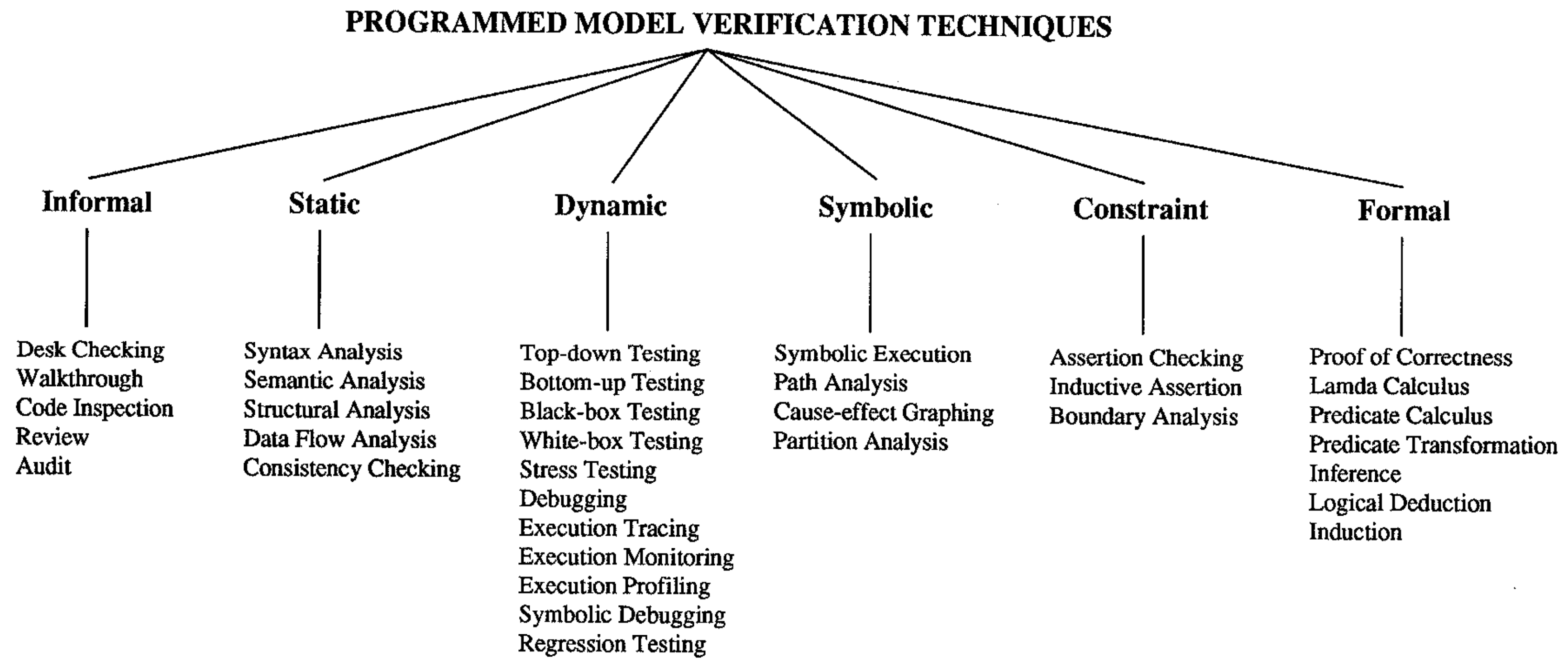

Figure 1. A Taxonomy for Programmed Model Verification Techniques

Table 1. Characteristics of the PMV Techniques Under Each Category

	Informal Analysis	**Static Analysis**	**Dynamic Analysis**	**Symbolic Analysis**	**Constraint Analysis**	**Formal Analysis**
Category Definition	Analyzing through the employment of informal 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

By applying verification techniques categorically, the modeler not only realizes a verified model, but also 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. TECHNIQUES FOR PROGRAMMED MODEL VERIFICATION

In this section, 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.

3.1 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.

3.1.1 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. It 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.

3.1.2 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] and 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: (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.

3.1.3 Code Inspection

Code inspections were introduced by 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 than 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 precede testing activities.

Fagan [1976] 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 have 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.

3.1.4 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.

3.1.5 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.

3.1.6 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 accurate 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.

3.2 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 simulation 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 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.

3.2.1 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.

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.

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. 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.

3.2.2 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's 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's intentions. It therefore becomes beneficial, even to the point of being essential, to tell the modeler what it is that he has specified 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, may be 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.

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 [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.

3.2.3 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.

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. 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. 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."

3.2.4 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. 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].

3.2.5 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 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.

3.2.6 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. 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.

3.3 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.

3.3.1 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.

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 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 than to let its caller complete the call. Fairley [1976] lists the following advantages of top-down testing: (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 which is discussed below.

3.3.2 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 is 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.

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.

3.3.3 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].

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.

3.3.4 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.

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.

3.3.5 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 3.5.3), 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. 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.

3.3.6 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.

3.3.7 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.

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.

3.3.8 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 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.

3.3.9 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 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.

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.

3.3.10 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. 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.

3.3.11 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 must 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.

3.3.12 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 very 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.

3.4 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.

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 may become 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.

3.4.1 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.

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 and depict the symbolic output produced.

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.

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]. 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]. Symbolic execution is also limited in its use with complex data structures because of difficulties in symbolically representing particular data elements within 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.

3.4.2 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. 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. 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.

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

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 depends 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.

3.4.4 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) 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.

3.4.5 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.

3.5 Constraint Analysis

Early in the Simulation Model Development Life Cycle [Balci 1989], 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 Conceptual Model (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.

3.5.1 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. 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. 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), 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).

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.

3.5.2 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.

3.5.3 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 $0 \leq rn \leq 1$. In effect, the model contains three separate equivalence partitions here: $0 \leq rn \leq 0.2$, $0.2 < rn \leq 0.55$, and $0.55 < rn \leq 1$. 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]. 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 3.3.5).

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.

3.5.4 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 responds 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.

3.6 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: (1) proof of correctness, (2) predicate calculus, (3) predicate transformation, (4) λ-calculus, (5) inference, (6) logical deduction, and (7) 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 manipulated 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 1977].

3.6.1 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.

4. CONCLUDING REMARKS

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 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 adequate 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 paper fills those voids in a number of ways, the first by contributing the Taxonomy for Programmed Model Verification Techniques. 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. Even the many software developers who have yet to understand the software verification process will find these resources invaluable.

The emphasis of this paper is on performing PMV in an effort to increase model credibility. This paper 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 13 CASs in the simulation model life cycle. It cannot be emphasized enough that the other aspects of the life cycle must not be overlooked.

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 paper 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.

REFERENCES

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," In *Proceedings of the Thirteenth Asilomar Conference on Circuits, Systems, and Computers*, Pacific Grove, Calif., pp. 302-305.

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. (1989), "How to Assess the Acceptability and Credibility of Simulation Results," In *Proceedings of the 1989 Winter Simulation Conference* (Washington, D.C., Dec. 4-6). IEEE, Piscataway, N.J., to appear.

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

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.

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 (Dar mstadt, FRG, Sept. 25-30), Hans-Ludwig Hausen, Ed., pp. 41-56.

Chow, R.S. (1976), "A Generalized Assertion Language," In *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 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.

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

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

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.

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.

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

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,"In *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.

Floyd, R.W. (1967), "Assigning Meaning to Programs," *Proceedings of the American Mathematical Society Symposium in Applied Mathematics 19*, pp. 19-31.

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.

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.

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.

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

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.

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.

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

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.

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

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

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

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.

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

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.

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

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.

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.

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.

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

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

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, Ed., 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," In *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.

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

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

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