

V&V Techniques

RPG Reference Document

8/15/01¹

Table of Contents

<u>V&V Technique Taxonomy</u>	1
<u>Informal V&V Techniques</u>	3
Audit	3
Desk Checking / Self-inspection	3
Face Validation	4
Inspection	4
Review	5
Turing Test	6
Walkthroughs	6
Inspection vs Walkthrough vs Review	6
<u>Static V&V Techniques</u>	7
Cause-Effect Graphing	7
Control Analysis	8
Data Analysis	9
Fault/Failure Analysis	9
Interface Analysis	9
Semantic Analysis	10
Structural Analysis	10
Symbolic Evaluation	11
Syntax Analysis	12
Traceability Assessment	12
<u>Dynamic V&V Techniques</u>	12
Acceptance Testing	13
Alpha Testing	13
Assertion Checking	13
Beta Testing	14

¹ This document replaces the 11/30/00 version. It contains formatting and minor editorial changes.

Table of Contents (continued)

Dynamic Testing Techniques (continued)

Bottom-Up Testing	14
Comparison Testing	15
Compliance Testing	15
Debugging	16
Execution Testing	16
Fault/Failure Insertion Testing	17
Field Testing	17
Functional Testing	17
Graphical Comparison	18
Interface Testing	18
Object-Flow Testing	20
Partition Testing	20
Predictive Validation	20
Product Testing	21
Regression Testing	21
Sensitivity Analysis	21
Special Input Testing	21
Statistical Techniques	23
Structural Testing	25
Submodel/Module Testing	27
Symbolic Debugging	27
Top-Down Testing	27
Visualization/Animation	28
<u>Formal Techniques</u>	28
<u>Guidelines for Selecting V&V Techniques</u>	30
<u>References</u>	36
RPG References in this Document	41
Additional References	41
<u>Appendix A: Validation Procedure Using Simultaneous Confidence Intervals</u>	A-1
<u>Appendix B: Selecting V&V Techniques for Defect Detection</u>	B-1

V&V Technique Taxonomy

This document describes over seventy-five V&V techniques and eighteen statistical techniques that can be used for model validation. Most of these techniques are derived from software engineering; the remaining are specific to the modeling and simulation (M&S) field. The software V&V techniques applicable to M&S V&V are presented in terms understandable by an M&S technical person. Some software V&V techniques have been modified for use in M&S V&V. The term **testing** is used frequently when referring to the implementation of these techniques because V&V involves the testing of the model or simulation to assess its credibility.

The V&V techniques discussed in this document are separated into four categories: informal, static, dynamic, and formal.

- **Informal V&V techniques** are among the most commonly used. They are called informal because their tools and approaches rely heavily on human reasoning and subjectivity without stringent mathematical formalism.
- **Static V&V techniques** assess the accuracy of the static model design and source code. Static techniques do not require machine execution of the model, but mental execution can be used. The techniques are very popular and widely used, and many automated tools are available to assist in the V&V process. Static techniques can reveal a variety of information about the structure of the model, the modeling techniques used, data and control flow within the model, and syntactical accuracy (Whitner and Balci, 1989).
- **Dynamic V&V techniques** require model execution; they evaluate the model based on its execution behavior. Most dynamic V&V techniques require **model instrumentation**, the insertion of additional code (probes or stubs) into the executable model to collect information about model behavior during execution. Dynamic V&V techniques usually are applied in three steps:
 - executable model is instrumented
 - instrumented model is executed
 - model output is analyzed and dynamic model behavior is evaluated
- **Formal V&V techniques (or formal methods)** are based on formal mathematical proofs or correctness and are the most thorough means of model V&V. The successful application of formal methods requires the model development process to be well defined and structured. Formal methods should be applied early in the model development process to achieve maximum benefit. Because formal techniques require significant effort they are best applied to complex problems, which cannot be handled by simpler methods.

Although these categories share many of the same characteristics and individual V&V techniques can overlap from one to another, the complexity and the mathematical and logical formalism involved increase as the category becomes more formal.

The taxonomy table below lists all the techniques discussed. They are grouped according to the categories described above and hot links are provided so the reader can select by either category or technique. The categories are also identified by color-coding.

Verification and Validation Technique Taxonomy			
<u>Informal Techniques</u>			
<u>audit</u>	<u>desk check</u>	<u>face validation</u>	<u>inspection</u>
<u>review</u>	<u>Turing test</u>	<u>walkthrough</u>	
<u>Static Techniques</u>			
<u>cause-effect graphing</u>	<u>control analyses</u> calling structure	control flow	<u>fault/failure analysis</u>
	concurrent process	state transition	
<u>interface analyses</u> model interface user interface		<u>semantic analysis</u>	<u>structural analysis</u>
		<u>syntax analysis</u>	<u>traceability assessment</u>
<u>Dynamic Techniques</u>			
<u>acceptance test</u>	<u>alpha test</u>	<u>assertion check</u>	<u>beta test</u>
<u>bottom-up test</u>	<u>comparison test</u>	<u>compliance tests</u> authorization security	<u>debugging</u>
		performance standards	
<u>execution tests</u> monitor profile trace	<u>fault / failure insertion test</u>	<u>field test</u>	<u>functional test (Black Box test)</u>
<u>graphical comparison</u>	<u>interface tests</u> data model user	<u>object-flow test</u>	<u>partition test</u>
<u>predictive validation</u>	<u>product test</u>	<u>regression test</u>	<u>sensitivity analysis</u>
<u>special input tests</u>		<u>structural tests (White Box tests)</u>	<u>statistical techniques</u>
boundary value	real-time input	branch loop	<u>submodel / module test</u>
equivalence partitioning	self-driven input	condition path	
extreme input	stress	data flow statement	
invalid input	trace-driven input		
<u>symbolic debugging</u>	<u>top-down test</u>	<u>visualization / animation</u>	
<u>Formal Techniques</u>			
<u>induction</u>	<u>inference</u>	<u>logical deduction</u>	<u>inductive assertion</u>
<u>lambda calculus</u>	<u>predicate calculus</u>	<u>predicate transformation</u>	<u>proof of correctness</u>

Informal V&V Techniques

Informal techniques are among the most commonly used. They are called informal because they rely heavily on human reasoning and subjectivity without stringent mathematical formalism. The **informal** label should not imply, however, a lack of structure or formal guidelines in their use. In fact, these techniques should be applied using well-structured approaches under formal guidelines. They can be very effective if employed properly.

Audit

An audit is a verification technique performed throughout the development life cycle of a new model or simulation or during modification made to legacy models and simulations. An audit is a staff function that serves as the "eyes and ears of management" [Perry, 1995, p. 26]. An audit is undertaken to assess how adequately a model or simulation is used with respect to established plans, policies, procedures, standards, and guidelines. Auditing is carried out by holding meetings and conducting observations and examinations [Hollocker, 1987]. The process of documenting and retaining sufficient evidence about the substantiation of accuracy is called an **audit trail** [Perry, 1995]. Auditing can be used to establish traceability within the simulation. When an error is identified, it should be traceable to its source via its audit trail.

Desk Checking / Self-inspection

Desk checking, or self-inspection, is an intense examination of a working product or document to ensure its correctness, completeness, consistency, and clarity. It is particularly useful during requirements verification, design verification, and code verification. Desk checking can involve a number of different tasks, such as those listed in the table below [Beizer, 1990].

Typical Desk Checking Activities
<ul style="list-style-type: none">• syntax review• cross-reference examination• convention violation assessment• detailed comparison to specifications• code reading• control flowgraph analysis• path sensitizing

To be effective, desk checking should be conducted carefully and thoroughly, preferably by someone not involved in the actual development of the product or document, because it is usually difficult to see one's own errors [Adrion *et al.*, 1982].

Face Validation

The project team members, potential users of the model, and subject matter experts (SMEs) review simulation output (e.g., numerical results, animations, etc.) for reasonableness. They use their estimates and intuition to compare model and system behaviors subjectively under identical input conditions and judge whether the model and its results are reasonable [Hermann, 1967].

Informal Techniques Example 1:

Face validation was used in the development of a simulation of the U.S. Air Force (AF) manpower and personnel system to ensure it provided an adequate representation. The simulation was designed to provide AF policy analysts with a system-wide view of the effects of various proposed personnel policies. The simulation was executed under the baseline personnel policy and results shown to AF analysts and decision-makers who subsequently identified some discrepancies between the simulation results and perceived system behavior. Corrections were made and additional face validation evaluations were conducted until the simulation appeared to closely approximate current AF policy. The face validation exercise both demonstrated the validity of the simulation and improved its perceived credibility [Morrison]

Face validation is regularly cited in V&V efforts within the Department of Defense (DoD) M&S community. However, the term is commonly misused as a more general term and misapplied to other techniques involving visual reviews (e.g., inspection, desk check, review). Face validation is useful mostly as a preliminary approach to validation in the early stages of development. When a model is not mature or lacks a well-documented VV&A history, additional validation techniques may be required.

Inspection

Inspection is normally performed by a team that examines the product of a particular simulation development phase (e.g., M&S requirements definition, conceptual model development, M&S design). A team normally consists of four or five members, including a moderator or leader, a recorder, a reader (i.e., a representative of the Developer) who presents the material being inspected, the V&V Agent; and one or more appropriate subject matter experts (SMEs). Normally, an inspection consists of five phases: overview, preparation, inspection, rework, and follow-up [Schach, 1996].

Informal Techniques Example 2:

The team inspecting a simulation design might include a moderator; a recorder; a reader from the simulation design team who will explain the design process and answer questions about the design; a representative of the Developer who will be translating the design into an executable form; SMEs familiar with the requirements of the application, and the V&V Agent.

- **Overview** -- The simulation design team prepares a synopsis of the design. This and related documentation (e.g., problem definition and objectives, M&S requirements, inspection agenda) is distributed to all members of the inspection team.

- **Preparation** --The inspection team members individually review all the documentation provided. The success of the inspection rests heavily on the conscientiousness of the team members in their preparation.
- **Inspection** -- The moderator plans and chairs the inspection meeting. The reader presents the product and leads the team through the inspection process. The inspection team can be aided during the faultfinding process by a checklist of queries. The objective is to identify problems, not to correct them. At the end of the inspection the recorder prepares a report of the problems detected and submits it to the design team.
- **Rework** --The design team addresses each problem identified in the report, documenting all responses and corrections.
- **Follow-up** -- The moderator ensures that all faults and problems have been resolved satisfactorily. All changes should be examined carefully to ensure that no new problems have been introduced as a result of a correction.

Review

A review is intended to evaluate the simulation in light of development standards, guidelines, and specifications and to provide management, such as the User or M&S PM, with evidence that the simulation development process is being carried out according to the stated objectives. A review is similar to an inspection or walkthrough, except that the review team also includes management. As such, it is considered a higher-level technique than inspection or walkthrough.

A review team is generally comprised of management-level representatives of the User and M&S PM. Review agendas should focus less on technical issues and more on oversight than an inspection. The purpose is to evaluate the model or simulation relative to specifications and standards, recording defects and deficiencies. The V&V Agent should gather and distribute the documentation to all team members for examination before the review. The V&V Agent should also prepare a set of indicators to measure such as those listed in the table below.

Review Indicators
• appropriateness of the problem definition and M&S requirements
• adequacy of all underlying assumptions
• adherence to standards
• modeling methodology
• quality of simulation representations
• model structure
• model consistency
• model completeness
• documentation

The V&V Agent may also prepare a checklist to help the team focus on the key points. The result of the review should be a document recording the events of the meeting,

deficiencies identified, and review team recommendations. Appropriate actions should then be taken to correct any deficiencies and address all recommendations.

Turing Test

The Turing test is used to verify the accuracy of a simulation by focusing on differences between the system being simulated and the simulation of that system. System experts are presented with two blind sets of output data, one obtained from the model representing the system and one from the system, created under the same input conditions and are asked to differentiate between the two. If they cannot differentiate between the two, confidence in the model's validity is increased [Schruben, 1980; Turing, 1963; Van Horn, 1971]. If they can differentiate between them, they are asked to describe the differences. Their responses provide valuable feedback regarding the accuracy and appropriateness of the system representation.

Walkthroughs

The main thrust of the walkthrough is to detect and document faults; it is not a performance appraisal of the Developer. This point must be made to everyone involved so that full cooperation is achieved in discovering errors. A typical structured walkthrough team consists of

- Coordinator, often the V&V Agent, who organizes, moderates, and follows up the walkthrough activities
- Presenter, usually the Developer
- Recorder
- Maintenance oracle, who focuses on long-term implications
- Standards bearer, who assesses adherence to standards
- Accreditation Agent, who reflects the needs and concerns of the User
- Additional reviewers such as the M&S PM and auditors

Except for the Developer, none of the team members should be involved directly in the development effort. [Adrion *et al.*, 1982; Deutsch, 1982; Myers, 1978, 1979; Yourdon, 1985].

Inspection vs Walkthrough vs Review

Inspections differ significantly from walkthroughs. An inspection is a five-step, formalized process. The inspection team uses the checklist approach for uncovering errors. A walkthrough is less formal, has fewer steps, and does not use a checklist to guide or a written report to document the team's work. Although the inspection process takes much longer than a walkthrough, the extra time is justified because an inspection is extremely effective for detecting faults early in the development process when they

are easiest and least costly to correct [Ackerman *et al.*, 1983; Beizer, 1990; Dobbins, 1987; Knight and Myers, 1993; Perry, 1995; Schach, 1996].

Inspections and walkthroughs concentrate on assessing correctness. Reviews seek to ascertain that tolerable levels of quality are being attained. The review team is more concerned with design deficiencies and deviations from the conceptual model and M&S requirements than it is with the intricate line-by-line details of the implementation. The focus of a review is not on discovering technical flaws but on ensuring that the design and development fully and accurately address the needs of the application. For this reason, the review process is effective early on during requirements verification and conceptual model validation. [Hollocker, 1987; Perry, 1995; Sommerville, 1996; Whitner and Balci, 1989].

Static V&V Techniques

Static V&V techniques assess the accuracy of the static model design and source code. They can reveal a variety of information about the structure of the model, modeling techniques used, data and control flows within the model, and syntactical accuracy [Whitner and Balci, 1989]. Static techniques do not require machine execution of the model but mental execution or rehearsal is often involved. Static V&V techniques are widely used and many automated tools are available. For example, the simulation language compiler is itself a static V&V tool.

Cause-Effect Graphing

Cause-effect graphing addresses the question of what causes what in the model representation. Causes and effects are first identified in the system being modeled and then their representations are examined in the model specification.

Static Techniques Example 1:

In the simulation of a traffic intersection, the following causes and effects may be identified:

- the change of a light to red immediately causes the vehicles in the traffic lane to stop
- an increase in the duration of a green light causes a decrease in the average waiting time of vehicles in the traffic lane
- an increase in the arrival rate of vehicles causes an increase in the average number of vehicles at the intersection

As many causes and effects as possible should be listed. The semantics are expressed in a cause-effect graph that is annotated to describe special conditions or impossible situations. Once the cause-effect graph has been constructed, a decision table is created by tracing back through the graph to determine combinations of causes that result in each effect. The decision table then is converted into test cases with which the model is tested [Myers, 1979; Pressman, 1996; Whitner and Balci, 1989].

Control Analysis

Control analysis techniques include calling structure analysis, concurrent process analysis, control flow analysis, and state transition analysis.

Calling structure analysis is used to assess model accuracy by identifying **who calls whom** and **who is called by whom**. The **who** can be a procedure, subroutine, function, method, or a submodel within a model.

Static Techniques Example 2:

Inaccuracies caused by message passing (e.g., sending a message to a nonexistent object) in an object-oriented model can be revealed by analyzing the specific messages that invoke an action and the actions that messages invoke [Miller *et al.*, 1995].

Concurrent process analysis is especially useful for parallel [Fujimoto, 1990, 1993; Page and Nance, 1994] and distributed simulations.

- a simulation executing on a single computer with a single processor (CPU) is referred to as a **serial (sequential) simulation**
- a simulation executing on a single computer with multiple processors is a **parallel simulation**
- a simulation executed on multiple single-processor computers is said to be a **distributed simulation**

Model accuracy is assessed by analyzing the overlap or simultaneous execution of actions executed in parallel or across distributed simulations. Such analysis can reveal synchronization and time management problems [Ratray, 1990].

Control flow analysis examines sequences of control transfers and is useful for identifying incorrect or inefficient constructs within model representation. A graph of the model is constructed in which conditional branches and model junctions are represented by nodes and model segments between such nodes are represented by links [Beizer, 1990]. A node of the model graph usually represents a logical junction where the flow of control changes, whereas an edge represents the junction that assumes control.

State transition analysis identifies the finite number of states through which the model execution passes. A state transition diagram is used to show how the model transitions from one state to another. Model accuracy is assessed by analyzing the conditions under which a state change occurs. This technique is especially effective for models and simulations created under activity scanning, three-phase, and process interaction conceptual frameworks [Balci, 1988].

Data Analysis

Data analysis techniques are used in V&V activities to ensure that

- proper operations are applied to data objects (e.g., data structures, event lists, linked lists)
- data used by the model are properly defined
- defined data are properly used [Perry, 1995]

Two basic data analysis techniques are data dependency analysis and data flow analysis.

Data dependency analysis determines which variables depend on other variables [Dunn, 1984]. For parallel and distributed simulations, the data dependency knowledge is critical for assessing the accuracy of synchronization across multiple processors.

Data flow analysis assesses model accuracy with respect to the use of model variables. This assessment 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 flowgraph is 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 or federates [Allen and Cocke, 1976; Whitner and Balci, 1989].

Fault/Failure Analysis

Fault (i.e., incorrect model component) and failure (i.e., incorrect behavior of a model component) analysis uses model input-output transformation descriptions to identify how the model logically might fail. The model design specification is examined to determine if any failures logically could occur, in what context, and under what conditions. Such examinations often lead to identification of model defects [Miller *et al.*, 1995].

Interface Analysis

Interface analysis techniques are especially useful for verification and validation of interactive and distributed simulations. Two basic techniques are model interface analysis and user interface analysis.

- **Model interface analysis** examines submodel-to-submodel interfaces within a model, or federate-to-federate interfaces within a federation, and determines if the interface structure and behavior are sufficiently accurate.
- **User interface analysis** examines the user-model interface and determines if it is human engineered to prevent errors during the user's interactions with the model. It also assesses how accurately this interface is integrated into the overall model or simulation.

Semantic Analysis

Semantic analysis is conducted by the simulation programming language compiler and determines the modeler's intent as reflected by the code. The compiler describes the content of the source code so the modeler can verify that the original intent is reflected accurately.

The compiler generates a wealth of information to help the modeler determine if the true intent is translated accurately into the executable code, such as

- **symbol tables**, which describe the elements or symbols that are manipulated in the model, function declarations, type and variable declarations, scoping relationships, interfaces, and dependencies
- **cross-reference tables**, which describe called versus calling routines (where each data element is declared, referenced, and altered), duplicate data declarations (how often and where occurring), and unreferenced source code
- **subroutine interface tables**, which describe the actual interfaces of the caller and the called
- **maps**, which relate the generated runtime code to the original source code
- **pretty printers or source code formatters**, which reformat the source listing on the basis of its syntax and semantics, clean pagination, highlighting of data elements, and marking of nested control structures [Whitner and Balci, 1989]

Structural Analysis

Structural analysis is used to examine the model structure and determine if it adheres to structure principles. It is conducted by constructing a control flowgraph of the model structure and examining the graph 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 (e.g., GOTOs).

Yucesan and Jacobson (1992, 1996) apply the theory of computational complexity and show that the problem of verifying structural properties of M&S applications is difficult to solve. They illustrate that modeling issues such as accessibility of states, ordering of events, ambiguity of model specifications, and execution stalling are problems for which general design techniques do not produce efficient solutions.

Symbolic Evaluation

Symbolic evaluation assesses model accuracy by exercising the model using symbolic values rather than actual data values for input. It is performed by feeding symbolic inputs into a component or federate and producing expressions for the output that are derived from the transformation of the symbolic data along model execution paths.

Static Techniques Example 3:

```
function jobArrivalTime(arrivalRate,currentClock,randomNumber)
  lag = -10
  Y = lag * currentClock
  Z = 3 * Y
  if Z < 0 then
    arrivalTime = currentClock - log(randomNumber) / arrivalRate
  else
    arrivalTime = Z - log(randomNumber) / arrivalRate
  end if
  return arrivalTime
end jobArrivalTime
```

In symbolic execution, lag is substituted in Y resulting in $Y = (-10 * \text{currentClock})$. Substituting again, Z is found to be equal to $(-30 * \text{currentClock})$. Since currentClock is always zero or positive, an error is detected in that Z will never be greater than zero, and the "if-then-else" statement is unnecessary.

When unresolved conditional branches are encountered, a path is chosen to traverse. Once a path is selected, execution continues down the new path. At some point, 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 [Adrion et al., 1982; King, 1976]. 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.

Symbolic evaluation assists in showing path correctness for all computations regardless of test data and is also a great source of documentation, but it has the following disadvantages [Dillon, 1990; King, 1976; Ramamoorthy et al., 1976]:

- the execution tree can explode in size and become too complex as the model grows
- loops cause difficulties although inductive reasoning and constraint analysis may help
- loops make thorough execution impossible because all paths must be traversed
- complex data structures may have to be excluded because of difficulties in symbolically representing particular data elements within the structure

Syntax Analysis

Syntax analysis is done by the simulation programming language compiler to ensure that the mechanics of the language are applied correctly [Beizer, 1990].

Traceability Assessment

Traceability assessment is used to match the individual elements of one form of the model to another. For example, the elements of the system as described in the requirements specification are matched one to one to the elements of the simulation design specification. Unmatched elements *may* reveal either unfulfilled requirements or unintended design functions [Miller *et al.*, 1995].

Dynamic V&V Techniques

Dynamic V&V techniques evaluate the model based on its execution behavior and as such require model execution. Most dynamic V&V techniques require **model instrumentation**, the insertion of additional code (probes or stubs) into the executable model to collect information about model behavior during execution. Probe locations are determined manually or automatically based on static analysis of the model's structure. Automated instrumentation is accomplished by a preprocessor that analyzes the model's static structure (usually via graph-based analysis) and inserts probes at appropriate places.

Dynamic V&V techniques usually are applied in three steps:

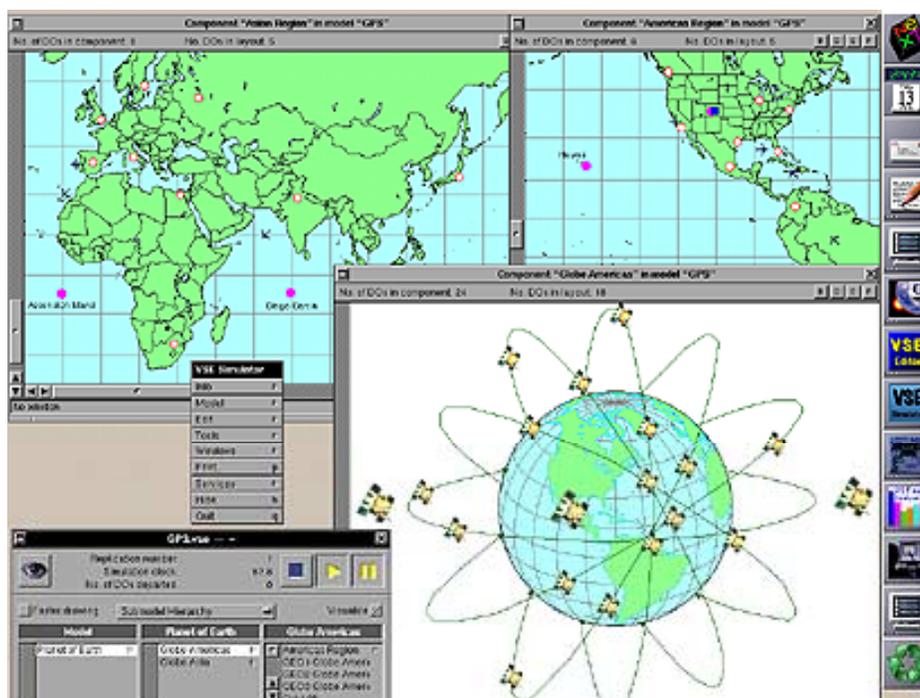
- 1) executable model is instrumented
- 2) instrumented model is executed
- 3) model output is analyzed and dynamic model behavior is evaluated

Dynamic Techniques Example 1:

Consider a worldwide air traffic control and satellite communication object-oriented visual M&S application created by using the Visual Simulation Environment [Balci *et al.*, 1995]. In step 1, the model is instrumented to record the following information every time an aircraft enters into the coverage area of a satellite:

- aircraft tail number
- time
- aircraft's longitude, latitude, and altitude
- satellite's position and identification number

In Step 2, the model is executed and the information collected is written to an output file. In Step 3, the output file is examined to reveal discrepancies and inaccuracies in model representation.



Visual Simulation of Global Air Traffic Control and Satellite Communication
(reprint from Balci, et al., 1995)

Acceptance Testing

Acceptance testing is conducted by either the M&S User and the V&V Agent or the Developer's quality control group in the presence of the User's representatives. The model is operationally tested with the actual hardware and data to determine whether all requirements specified in the legal contract are satisfied [Perry, 1995; Schach, 1996].

Alpha Testing

Alpha testing is the operational testing of the initial version of the complete model by the developer at an in-house site uninvolved with the model development [Beizer, 1990].

Assertion Checking

An assertion is a statement that should hold true as the simulation executes. Assertion checking is a verification technique that checks what is happening against what the modeler assumes is happening to guard against potential errors. The assertions are placed in various parts of the model to monitor execution. They can be inserted to hold true globally, for the whole model; regionally, for some submodels; locally, within a submodel; or at entry and exit of a submodel.

Dynamic Techniques Example 2:

Consider the following pseudo-code [Whitner and Balci, 1989]:

```
Base := Hours * PayRate
```

```
Gross := Base * (1 + BonusRate)
```

In just these two simple statements, several assumptions are being made. It is assumed that Hours, PayRate, Base, BonusRate, and Gross are all non-negative. The following asserted code can be used to prevent execution errors caused by incorrect values entered by the user:

```
Assert Local (Hours > 0 and PayRate > 0 and BonusRate > 0)
```

```
Base := Hours * PayRate
```

```
Gross := Base * (1 + BonusRate)
```

Assertion checking also prevents structural model inaccuracies. For example, the model discussed in [dynamic techniques example 1](#) can contain assertions such as

- a satellite communicates with the correct ground station
- an aircraft's tail number matches its type
- an aircraft's flight path is consistent with the official airline guide

Clearly, assertion checking serves two important needs:

- it verifies that the model is functioning within its acceptable domain
- assertion statement documents the intentions of the modeler

Assertion checking, however, degrades model performance, forcing the modeler to choose between execution efficiency and accuracy. If the execution performance is critical, the assertions should be turned off but kept permanently in code to provide both documentation and means for maintenance testing [Adrion *et al.*, 1982].

Beta Testing

Beta testing refers to the developer's operational testing of the first-release version of the complete model at a beta user site under realistic field conditions [Miller *et al.*, 1995].

Bottom-Up Testing

Bottom-up testing is used with bottom-up model development. Many well-structured models consist of a hierarchy of submodels. In bottom-up development, model construction starts with the simulation's routines at the base level, i.e., the ones that cannot be decomposed further, and culminates with the submodels at the highest level. As each routine is completed, it is tested thoroughly. When routines with the same

parent, or submodel, have been developed and tested, the routines are integrated and their integration is tested. This process is repeated until all submodels and the model as a whole have been integrated and tested. The integration of completed submodels need not wait for all submodels at the same level to be completed. Submodel integration and testing can be, and often is, performed incrementally [Sommerville, 1996].

Some of the advantages of bottom-up testing include

- it encourages extensive testing at the routine and submodel levels
- the smaller the submodels and the more cohesion within the model, the easier and more complete its testing will be
- it is particularly attractive for testing distributed models and simulations.

A major disadvantage of bottom-up testing involves the need for test harnesses or drivers to simulate calling each submodel and to pass the test data needed to execute each submodel. Developing harnesses for every submodel can be quite complex and difficult and the harnesses themselves may contain errors. In addition, bottom-up testing faces the same cost and complexity problems as [top-down testing](#).

Comparison Testing

Comparison testing (also known as back-to-back testing) may be used when more than one version of a model or simulation is available for testing [Pressman, 1996; Sommerville, 1996]. For example, different simulations may have been developed by the different Services to simulate the same military combat aircraft. All simulations built to represent exactly the same system are run with the same input data and the model outputs are compared. Differences in the outputs reveal problems with model accuracy. The major disadvantage to this technique is the lack of information that generally exists about the validity of the other models. In addition, if two models were written with the same specific, unnoticed error, the results might agree but would still be invalid.

Compliance Testing

Compliance testing compares the simulation to required security and performance standards. These techniques are particularly useful for testing federations of distributed and interactive models and simulations. A number of different tests are involved:

- **Authorization testing** tests how accurately different levels of security access authorization are implemented in the simulation and how properly they comply with established rules and regulations. The test can be conducted by attempting to execute a classified model within a federation or by using classified input data to run a simulation without proper authorization [Perry, 1995].

- **Performance testing** simply tests whether all performance characteristics are measured and evaluated with sufficient accuracy and if all established performance requirements are satisfied [Perry, 1995].
- **Security testing** tests whether all security procedures are implemented correctly and properly. Security testing evaluates the adequacy of protective procedures and countermeasures by such methods as attempting to penetrate the simulation while it is running and attempting to break into protected components (e.g., secure databases) [Perry, 1995].
- **Standards testing** substantiates that the simulation or federation is developed with respect to the required standards, procedures, and guidelines.

Debugging

Debugging is a four-step iterative process used to uncover and correct errors and misconceptions that cause a model's failure.

- model is tested, revealing the existence of errors (bugs)
- cause of each detected error is determined
- model changes necessary to correct the detected errors are identified
- necessary model changes are made

The model should be retested immediately after changes are made to ensure successful modification, because a change correcting an error may create another one. This iterative process continues until no errors are identified after testing [Dunn, 1987].

Execution Testing

Execution testing includes monitoring, profiling, and tracing techniques. These techniques collect and analyze execution behavior data to reveal model representation errors.

- **Execution monitoring** reveals errors by examining low-level information about activities and events that take place during model execution. It requires the instrumentation of a model or simulation to gather data to provide activity- or event-oriented information about the model's dynamic behavior.
- **Execution profiling** reveals errors by examining high-level information (profiles) about activities and events that take place during model execution. It requires the instrumentation of an executable model to gather data to present profiles about the model's dynamic behavior.
- **Execution tracing** reveals errors by reviewing the line-by-line execution of a simulation. It requires the instrumentation of an executable model to trace the model's line-by-line dynamic behavior. The major disadvantage of the tracing technique is that execution of the instrumented model may produce a large volume of trace data too complex to analyze. To overcome this problem, the

trace data can be stored in a database and the modeler can analyze it using a query language [Fairley, 1975, 1976].

Dynamic Techniques Example 3:

The model in [dynamic techniques example 1](#) can be instrumented

- to monitor the arrivals and departures of aircraft within a particular city, and the results can be compared with the official airline guide to judge model validity. The model also can be instrumented to provide other low-level information such as the number of late arrivals, the average passenger waiting time at the airport, and the average flight time between two locations.
- to produce histograms of aircraft departure times, arrival times, and passenger checkout times at an airport.
- to record all aircraft arrival times at a particular airport. Then the trace data can be compared with the official airline guide to assess model validity.

Fault/Failure Insertion Testing

This technique inserts a fault (incorrect model component) or a failure (incorrect behavior of a model component) into the model and observes whether the model produces the invalid behavior as expected. Unexplained behavior may reveal errors in model representation.

Field Testing

Field testing places the model in an operational situation and collects as much information as possible for validation. Field testing conducted as part of the test and evaluation (T&E) process is particularly important in DoD system acquisition. Although it is usually difficult, expensive, and sometimes impossible to devise meaningful field tests for complex systems, their use wherever possible helps both the project team and decision makers develop confidence in the model [Shannon, 1975; Van Horn, 1971]. The greatest disadvantage of field testing is the lack of adequate test resources to produce statistically significant results. Often, simulation runs augment live test data in the development and decision processes.

Functional Testing

Functional testing (also called **black-box testing**) assesses the accuracy of model input-output transformation. It is applied by inputting test data to the model and evaluating the accuracy of the corresponding outputs. It is virtually impossible to test all input-output transformation paths for a reasonably large and complex simulation because the paths could number in the millions. Therefore, the objective of functional testing is to increase confidence in model input-output transformation accuracy as much as possible rather than to claim absolute correctness.

Generating test data is a crucially important but very difficult task. The law of large numbers does not apply. Successfully testing the model under 1,000 input values (i.e., test data) does not imply high confidence in model input-output transformation accuracy just because the number appears large. Instead, the number of input values used should be compared with the number of allowable input values to determine the percentage of the model input domain that is covered in testing. The more the model input domain is covered in testing, the more confidence is gained in the accuracy of the model input-output transformation [Howden, 1980; Myers, 1979].

Graphical Comparison

Graphical comparison is a subjective and heuristic but practical approach for examining the representational quality of variables. The graphs of values of model variables over time are compared with the graphs of values of system variables to investigate characteristics such as similarities in periodicity, skew, number, and location of inflection points; logarithmic rise and linearity; phase shift; trend lines; and exponential growth constants [Cohen and Cyert, 1961; Forrester, 1961; Miller, 1975; Wright, 1972].

Interface Testing

Interface testing (also known as **integration testing**) tests the data, model, and user interfaces. Interface testing is more rigorous than the [interface analysis](#).

Data Interface Testing -- This form of testing is used to assess the accuracy of data entered into the model or derived from the model during execution. All data interfaces are examined to substantiate that all aspects of data input and output are correct. This form of testing is particularly important for those simulations in which the inputs are read from a database or the results are stored in a database for later analysis. The model's interface to the database is examined to ensure correct importing and exporting of data [Miller *et al.*, 1995]. Data interface testing is key to the relationship between the VV&A effort and the corresponding data V&V effort.

Model Interface Testing -- This form of testing is used to detect model representation errors created as a result of component-to-component or federate-to-federate interface errors or invalid assumptions about the interfaces. It is essential that each submodel within a model or model (federate) within a federation is tested individually and found to be sufficiently accurate before model interface testing begins.

This procedure focuses on how well the submodels (or federates) are integrated with each other and is particularly useful for object-oriented and distributed simulations.

Object-Oriented Object Paradigm
<ul style="list-style-type: none">• created with public and private interfaces• interface with other objects through message passing• reused with their interfaces• inherit the interfaces and services of other objects.

Model interface testing assesses the accuracy of four types of interfaces, as identified by Sommerville (1996):

- **Parameter interfaces** that pass data or function references from one object to another
- **Shared memory interfaces** that enable objects to share a block of memory in which data are placed by one object and from which they are retrieved by other objects
- **Procedural interfaces** that implement the concept of encapsulation under the object-oriented paradigm—an object provides a set of services (procedures) that can be used by other objects and hides (encapsulates) the way a service is provided from the outside world
- **Message-passing interfaces** that enable an object to request the service of another object through message passing

Sommerville (1996) classifies interface errors into three categories:

- **Interface misuse** occurs when an object calls another and incorrectly uses its interface. For objects with parameter interfaces, a parameter may be of the wrong type or may be passed in the wrong order, or the wrong number of parameters may be passed.
- **Interface misunderstanding** occurs when object A calls object B without satisfying the underlying assumptions of object B's interface.

Dynamic Techniques Example 4:

Object A calls a binary search routine by passing an unordered list to be searched, when in fact the binary algorithm assumes that the list is already sorted.

- **Timing errors** occur in real-time, parallel, and distributed simulations that use a shared memory or a message-passing interface.

User Interface Testing -- This form of testing is used to assess the interactions between the User and the simulation and to detect model representation errors created as a result of user-model interface errors or invalid assumptions about the user interface. It is particularly important for testing human-in-the-loop and interactive simulations. The user interface is examined from low-level ergonomic aspects to instrumentation and controls and from human factors to global considerations of usability and appropriateness to identify potential errors [Miller *et al.*, 1995; Pressman, 1996; Schach, 1996].

Object-Flow Testing

Object-flow testing is similar to **transaction-flow testing** [Beizer, 1990] and **thread testing** [Sommerville, 1996]. It is used to assess model accuracy by exploring the life cycle of an object during model execution. Every time the dynamic object enters into a subroutine, the visualization of that subroutine is displayed. Every time the dynamic object interacts with another object within the subroutine, the interaction is highlighted. Examination of the way a dynamic object flows through the activities and processes and interacts with its environment during its lifetime in model execution is extremely useful for identifying errors in model behavior.

Dynamic Techniques Example 5:

A dynamic object (aircraft) can be marked for testing in the visual simulation environment for the model shown in [dynamic techniques example 1](#).

Partition Testing

Partition testing examines the model with the test data generated by analyzing the model's functional representations or partitions. It is accomplished by

- decomposing both the model specification and its implementation into functional representations (partitions)
- comparing the elements and prescribed functionality of each partition specification with the elements and actual functionality of the corresponding partition as it has been implemented in code
- deriving test data to test the functional behavior of each partition extensively
- testing the model with the generated test data

The model is decomposed or partitioned into functional representations (i.e., the model computations) through the use of [symbolic evaluation](#) techniques that maintain algebraic expressions of model elements and show model execution paths. Two computations are equivalent if they are defined for the same subset of the input domain that 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 show equivalence over a domain. When equivalence cannot be shown, partition testing is performed to locate errors or, as Richardson and Clarke (1985, p. 1488) state, to "increase confidence in the equality of the computations due to the lack of error manifestation." By involving both the model's specification and implementation, partition testing can provide more comprehensive test data coverage than other test data generation techniques.

Predictive Validation

Predictive validation is used to test the predictive ability of a model [Emshoff and Sisson, 1970]. It requires past input and output data from the system being modeled. The model is driven by past system input data and its outputs are compared with the corresponding past system output data. Predictive validation is often employed in Test and Evaluation (T&E) testing. It is also used in the Model-Test-Model development methodology, which uses the test data to make subsequent improvements to the model.

Product Testing

Successful testing of each component or federate does not guarantee overall simulation or federation credibility. Product testing, as well as [interface testing](#), can be performed to substantiate overall model credibility. Product testing is conducted by the Developer after all components have been successfully integrated (as demonstrated by the interface testing) and before the [acceptance testing](#) is performed by the User. Because no one wants the product (model) to fail the acceptance test, product testing should be conducted to ensure that all requirements specified in the legal contract are satisfied before the model is turned over to the User [Schach, 1996].

Regression Testing

Regression testing is used to investigate the relationships between variables and to ensure that corrections and modifications to the model do not create other errors or adverse side effects. Because the modified model is usually retested with the test data sets used previously, successful regression testing requires the retention and management of old test data sets throughout the model development life cycle.

Sensitivity Analysis

Sensitivity analysis is performed by systematically changing the values of model input variables and parameters over some range of interest and observing the effect upon model behavior [Shannon, 1975]. Unexpected effects may reveal invalidity. The input values also can be changed to induce errors to determine the sensitivity of model behavior to such errors. Sensitivity analysis can identify those input variables and parameters to which model behavior is very sensitive. Model validity then can be enhanced by ensuring that those values are specified with sufficient accuracy [Hermann, 1967; Miller, 1974a,b; Van Horn, 1971].

Special Input Testing

Special input testing assesses model accuracy by subjecting the model to a variety of inputs. There are eight types of tests: boundary value, equivalence partitioning, extreme input, invalid input, real-time input, self-driven input, stress, and trace-driven input techniques.

- **Boundary value testing** is used to examine the model's accuracy by using test cases on the boundaries of input equivalence classes. A model's input domain usually can be divided into classes of input data (known as equivalence classes) that cause the model to function the same way.

Dynamic Techniques Example 6:

A traffic intersection model might specify the probability of left turn in a three-way turning lane as 0.2, the probability of right turn as 0.35, and the probability of traveling straight as 0.45. This probabilistic branching can be implemented by using a uniform random-number generator that produces numbers in the range $0 \leq rn \leq 1$. Thus, three equivalence classes are identified:

$$0 \leq rn \leq 0.2$$

$$0.2 < rn \leq 0.55$$

$$0.55 < rn \leq 1.$$

Each test case from within a given equivalence class has the same effect on the model behavior, i.e., produces the same direction of turn.

Because in boundary analysis, test cases are generated just within, on top of, and outside of the equivalence classes [Myers, 1979], for left turn, the following test cases should be selected: 0.0, ± 0.000001 , 0.199999, 0.2, and 0.200001.

In addition to generating test data on the basis of input equivalence classes, it also is useful to generate test data that 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 invalid test cases used in the example will cause the model execution to fail; however, this failure should be as expected and meaningfully documented.

- **Equivalence partitioning testing** partitions the model input domain into equivalence classes in such a manner that a test of a representative value from a class is assumed to be a test of all values in that class [Miller *et al.*, 1995; Perry, 1995; Pressman, 1996; Sommerville, 1996].
- **Extreme input testing** is conducted by running the model or simulation with only minimum values, maximum values, or an arbitrary mixture of minimum and maximum values for the model input variables. For example, this technique allows the model user to test a proposed weapon system against extreme conditions that may not be obtainable in actual system testing.
- **Invalid input testing** is performed by running the model or simulation under incorrect input data to determine whether the model behaves as expected. Unexplained behavior may reveal errors in model representations.
- **Real-time input testing** is particularly important for assessing the accuracy of simulations built to represent embedded real-time systems.

Dynamic Techniques Example 7:

Different design strategies of a real-time software system built to control the operations of a manufacturing system can be studied using simulation. The model that represents the software design can be tested by running it with real-time input data that can be collected from the existing manufacturing system.

Using real-time input data collected from a real system is particularly important to capture the timing relationships and correlations between input data points.

- **Self-driven input testing** is conducted by running the model or simulation under input data randomly sampled from probabilistic models representing random phenomena in a real or future system. A probability distribution (e.g., exponential, gamma, weibull) can be fit to collected data, or triangular and beta probability distributions can be used in the absence of data, to model random input conditions [Banks *et al.*, 1996; Law and Kelton, 1991]. Then, using random variate generation techniques, random values can be sampled from the probabilistic models to test the model validity under a set of observed or speculated random input conditions.
- **Stress testing** tests the model's validity under extreme workload conditions. This is usually accomplished by increasing the congestion in the model.

Dynamics Technique Example 6:

The model in dynamic techniques example 1 can be stress tested by increasing the number of flights between two locations to an extremely high value. Such an increase in workload may create unexpected high congestion in the model.

Under stress testing, the model may exhibit invalid behavior; however, such behavior should be as expected and meaningfully documented [Dunn, 1987; Myers, 1979].

- **Trace-driven input testing** is conducted by running the model or simulation under input trace data collected from a real system. A system can be instrumented with monitors that collect data by tracing all system events. The raw trace data can then be refined to produce the real input data for testing the model or simulation.

Statistical Techniques

Much research has been conducted in applying statistical techniques to model validation. The table below presents the statistical techniques proposed for model validation and lists related references.

Statistical Techniques Proposed for Validation	
Technique	References

Statistical Techniques Proposed for Validation	
Technique	References
Analysis of Variance	Naylor and Finger, 1967
Confidence Intervals/Regions	Balci and Sargent, 1984; Law and Kelton, 1991; Shannon, 1975
Factor Analysis	Cohen and Cyert, 1961
Hotelling's T ² Tests	Balci and Sargent, 1981, 1982a, 1982b, 1983; Shannon, 1975
Multivariate Analysis of Variance: <ul style="list-style-type: none"> – Standard MANOVA – Permutation Methods – Nonparametric Ranking Methods 	Garratt, 1974
Nonparametric Goodness-of-Fit Tests: <ul style="list-style-type: none"> – Kolmogorov-Smirnov Test – Cramer-Von Mises Test – Chi-square Test 	Gafarian and Walsh, 1969; Naylor and Finger, 1967
Nonparametric Tests of Means <ul style="list-style-type: none"> – Mann-Whitney-Wilcoxon Test – Analysis of Paired Observations 	Shannon, 1975
Regression Analysis	Aigner, 1972; Cohen and Cyert, 1961; Howrey and Kelejian, 1969
Theil's Inequality Coefficient	Kheir and Holmes, 1978; Rowland and Holmes, 1978; Theil, 1961
Time Series Analyses: <ul style="list-style-type: none"> – Spectral Analysis – Correlation Analysis – Error Analysis 	Fishman and Kiviat, 1967; Gallant et al., 1974; Howrey and Kelejian, 1969; Hunt, 1970; Van Horn, 1971; Watts, 1969 Watts, 1969 Damborg and Fuller, 1976; Tytula, 1978
t-Test	Shannon, 1975; Teorey, 1975

The statistical techniques listed in the table above require the system being modeled to be completely observable; i.e., all data required for model validation can be collected from the system. The model is validated by using the statistical techniques to compare the model output data with the corresponding system output data after the model is run with the same input data as the real system. Model and system outputs are compared using multivariate statistical techniques to capture the correlation among the output variables. A recommended validation procedure based on the use of simultaneous confidence intervals is provided in the [Appendix A](#).

Structural Testing

Structural testing (also called **white-box testing**) evaluates the model based on its internal structure (how it is built), whereas [functional \(black-box\) testing](#) assesses the input-output transformation accuracy of the model. Structural testing employs data flow and control flow diagrams to assess the accuracy of internal model structure by examining model elements such as statements, branches, conditions, loops, internal logic, internal data representations, submodel interfaces, and model execution paths.

Structural (white-box) testing consists of six testing techniques:

- branch testing
- condition testing
- data flow testing
- loop testing
- path testing
- statement testing

Branch Testing runs the model or simulation under test data to execute as many branch alternatives as possible, as many times as possible, and to substantiate their accurate operation. The more branches that test successfully, the more confidence is gained in the model's accurate execution with respect to its logical branches [Beizer, 1990].

Condition testing runs the model or simulation under test data to execute as many logical conditions as possible, as many times as possible, and to substantiate their accurate operation. The more logical conditions that test successfully, the more confidence is gained in the model's accurate execution with respect to its logical conditions.

Date flow testing uses the control flowgraph to explore sequences of events related to the status of data structures and to examine data-flow anomalies. For example, sufficient paths can be forced to execute under test data to ensure that every data element and structure is initialized before use or every declared data structure is used at least once in an executed path [Beizer, 1990].

Loop testing runs the model or simulation under test data to execute as many loop structures as possible, as many times as possible, and to substantiate their accurate operation. The more loop structures that test successfully, the more confidence is gained in the model's accurate execution with respect to its loop structures [Pressman, 1996].

Path testing runs the model or simulation under test data to execute as many control flow paths as possible, as many times as possible, and to substantiate their accurate operation. The more control flow paths that test successfully, the more confidence is

gained in the model's accurate execution with respect to its control flow paths, but 100 percent path coverage is impossible to achieve for a reasonably large M&S application [Beizer, 1990].

Path testing is performed in three steps [Howden, 1976].

- 1) The model control structure is determined and represented in a control flow diagram
- 2) Test data is generated to cause selected model logical paths to be executed. [Symbolic evaluation](#) can be used to identify and classify input data based on the symbolic representation of the model. The test data is generated in such a way as to
 - cover all statements in the path
 - encounter all nodes in the path
 - cover all branches from a node in the path
 - achieve all decision combinations at each branch point in the path
 - traverse all paths [Prather and Myers, 1987]
- 3) By using the generated test data, the model is forced to proceed through each path in its execution structure, thereby providing comprehensive testing.

In practice, only a subset of all possible model paths is selected for testing due to budgetary constraints. Recent work has sought to increase the amount of coverage per test case and to improve the effectiveness of the testing by selecting the most critical areas to test. The path prefix strategy is an adaptive strategy that uses previously tested paths 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 is a strategy that reduces the path coverage required by nearly 40 percent [Chusho, 1987] by eliminating nonessential paths. Paths overlapped by other paths are nonessential. The model control flow graph is transformed into a directed graph whose arcs (called **primitive arcs**) correspond to the essential paths of the model. Nonessential 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 (1987) presents algorithms for efficiently identifying nonessential paths, reducing the control graph into an inheritor-reduced graph, and applying the concept of essential paths to the selection of effective test data.

Statement testing runs the model or simulation under test data to execute as many statements as possible, as many times as possible, and to substantiate their accurate operation. The more statements that test successfully, the more confidence is gained in the model's accurate execution with respect to its statements [Beizer, 1990].

Submodel / Module Testing

Submodel testing requires a top-down decomposition of the model into submodels. The executable model is instrumented to collect data on all input and output variables of a submodel. The system is instrumented (if possible) to collect similar data. Then, the behavior of each submodel is compared with the corresponding subsystem's behavior to judge the submodel's validity. If a subsystem can be modeled analytically, its exact solution can be compared against the simulation solution to assess its validity quantitatively.

Validating each submodel individually does not imply sufficient validity for the whole model. Each submodel is found sufficiently valid with some allowable error. The allowable errors can accumulate to make the whole model invalid. Therefore, after each submodel is validated, the whole model itself must be tested.

Symbolic Debugging

This technique employs 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 interact with the entire model one step at a time, at predetermined locations, or under specified conditions. While using a symbolic debugger, the modeler may alter model data values or replay a portion of the model, i.e., execute it again under the same conditions. Typically, the modeler utilizes the information gathered with [execution testing techniques](#) to isolate a problem or its proximity. Then the debugger is employed to determine how and why the error occurs.

Current state-of-the-art debuggers can view the runtime code as it appears in the source listing, set watch variables to monitor data flow, examine complex data structures, and even communicate with asynchronous input/output channels. The use of symbolic debugging can reduce greatly the debugging effort while increasing its effectiveness. Symbolic debugging allows the modeler to locate errors and check numerous circumstances that lead to errors [Whitner and Balci, 1989].

Top-Down Testing

Top-down testing is used with top-down model development. In top-down development, model construction starts with the submodels at the highest level and culminates with the routines at the base level, i.e., the ones that cannot be decomposed further. As each submodel is completed, it is tested thoroughly. When submodels with the same parent have been developed and tested, the submodels are integrated and their integration is tested. This process is repeated until the whole model has been integrated and tested. The integration of completed submodels need not wait for all submodels at the same level to be completed. Submodel integration and testing can be, and often is, performed incrementally [Sommerville, 1996].

Top-down testing begins with a test of the global model at its highest level. When testing a given level, calls to submodels at lower levels are simulated using stubs. A

stub is a dummy submodel that has no function other than to let its caller complete the call. Fairley (1976) lists the following advantages of top-down testing:

- model integration testing is minimized
- a working model is produced earlier in the development process
- higher level interfaces are tested first
- a natural environment for testing lower levels is created
- errors are localized to new submodels and interfaces

Some of the disadvantages of top-down testing include [Fairley, 1976]:

- thorough submodel testing is discouraged, because the entire model must be executed to perform testing
- testing can be expensive, because the whole model must be executed for each test
- adequate input data is difficult to obtain because of the complexity of the data paths and control predicates
- integration testing is hampered because of the size and complexity of testing the whole model

Visualization / Animation

Visualization and animation of a simulation greatly assist in model V&V [Sargent, 1992]. Displaying graphical images of internal (e.g., how customers are served by a cashier) and external (e.g., utilization of the cashier) dynamic behavior of a model during execution exhibits errors.

Dynamic Techniques Example 8:

In visual simulation of a traffic intersection, the modeler can observe the arrival of vehicles in different lanes and their movements through the intersection as the traffic light changes. Visualizing the model as it executes and comparing it with the real traffic intersection can help identify discrepancies between the model and the system.

Seeing the model in action is very useful for uncovering errors; however, it does not guarantee model correctness [Paul, 1989]. Therefore, visualization should be used with caution.

Formal Techniques

Formal V&V techniques are based on formal mathematical proofs of correctness. If attainable, a formal proof of correctness is the most effective means of model V&V.

Unfortunately, “if attainable” is the sticking point. Current formal proof of correctness techniques cannot even be applied to a reasonably complex simulation; however, formal techniques can serve as the foundation for other V&V techniques. The most commonly known techniques are briefly described below [Khanna, 1991; Whitner and Balci, 1989].

Induction, Inference, and Logical Deduction 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**. Inductive reasoning is based on invariant properties of a set of observations; assertions are invariants because their value is defined to be true. Given that the initial model assertion is correct, it stands to reason that if each path progressing from that assertion is correct and each path subsequently progressing from the previous assertion is correct, then the model must be correct if it terminates. Birta and Ozmizrak (1996) present a knowledge-based approach for M&S validation that uses a validation knowledge base containing rules of inference.

Inductive Assertions assess model correctness based on an approach that is very close to formal proof of model correctness. It is conducted in three steps.

- 1) Input-to-output relations for all model variables are identified
- 2) These relations are converted into assertion statements and are placed along the model execution paths so that an assertion statement lies at the beginning and end of each model execution path
- 3) Verification is achieved by proving for each path that, 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 [Manna *et al.*, 1973; Reynolds and Yeh, 1976].

Lambda Calculus [Barendregt, 1981] is a system that transforms the model into formal expressions by rewriting strings. The model itself can be considered a large string. Lambda calculus specifies rules for rewriting strings to transform the model into lambda calculus expressions. Using lambda calculus, the modeler can express the model formally to apply mathematical proof of correctness techniques to it.

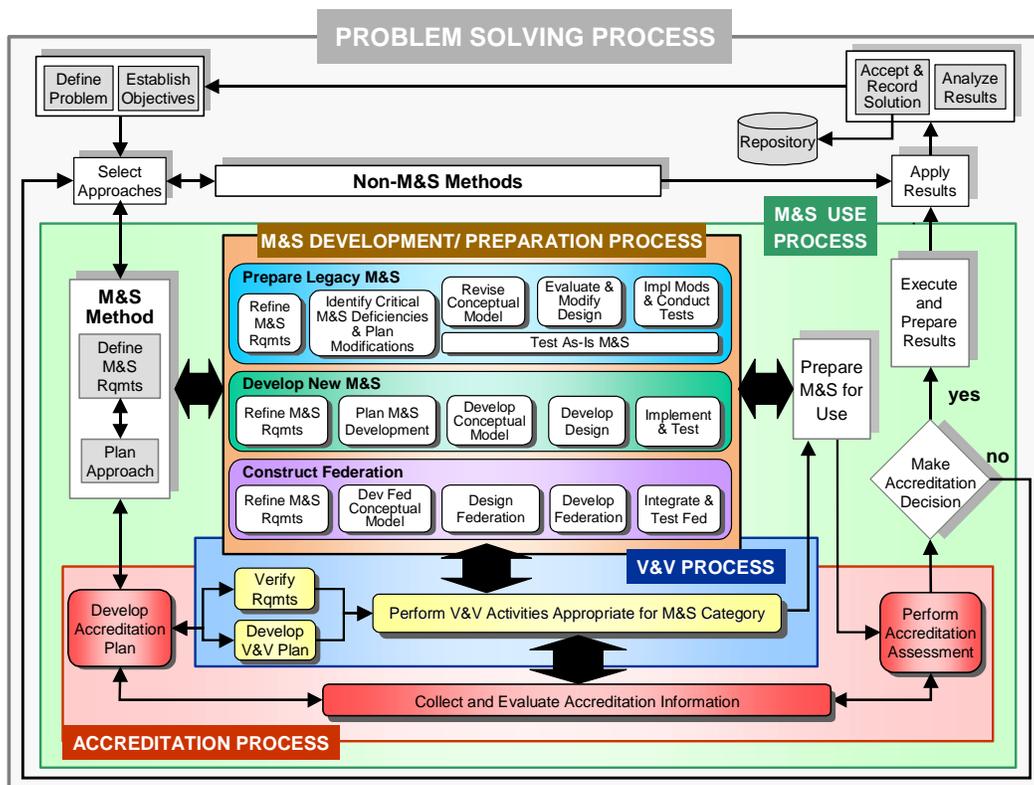
Predicate Calculus provides rules for manipulating predicates. A predicate is a combination of simple relations, such as **completed_jobs > steady_state_length**. A predicate will be either true or false. The model can be defined in terms of predicates and manipulated using the rules of predicate calculus. Predicate calculus forms the basis of all formal specification languages [Backhouse, 1986].

Predicate Transformation [Dijkstra, 1975; Yeh, 1977] verifies model correctness by formally defining the semantics of the model with a mapping that transforms model output states to all possible model input states. This representation is the basis from which model correctness is proved.

Formal **Proof of Correctness** expresses the model in a precise notation and then mathematically proves that the executed model terminates and satisfies the requirements with sufficient accuracy [Backhouse, 1986; Schach, 1996]. Attaining proof of correctness in a realistic sense is not possible under the current state of the art. The advantage of realizing proof of correctness is so great, however, that, when the capability is realized, it will revolutionize V&V.

Guidelines for Selecting V&V Techniques

In the overall problem solving process diagram shown below, the **V&V Process** is depicted as a subprocess of the **M&S Use Process** that interacts with both the **M&S Development/Preparation Process** and the **Accreditation Process**. (See the diagrams for VV&A and the New Development Process, VV&A and Legacy M&S Preparation, and VV&A and Federation Construction for a more detailed view of these interactions.)



The Overall Problem Solving Process

5/15/01

Any V&V process involves a series of activities and tasks that are selected to address the particular needs of the application and to map to the phases and activities of the particular development or preparation process involved. . What tasks are selected and what techniques are chosen to accomplish them depend upon a number of factors, such as

- type of simulation (legacy, new M&S, federation)
- problem to be solved
- objectives and requirements and their acceptability criteria
- risks and priorities of the User
- constraints (time, money, personnel, equipment)

See the core documents for V&V Agent Role in the VV&A of New Simulations, V&V Agent Role in the VV&A of Legacy Simulations, V&V Agent Role in the VV&A of Federations for additional information about specific V&V activities and tasks.

In the table below, the specific informal, static, dynamic, and formal V&V techniques, listed in the [V&V technique taxonomy table](#) and discussed in the first part of this document are mapped to the basic phases of simulation development and use (i.e., requirements definition, conceptual model development, design, implementation, use, and assessment). Brief synopses of the techniques are provided by hot link. Additional columns also indicate whether a technique is used primarily to support verification, validation, or both.. Selecting the best technique to apply to a given V&V task in a given situation is not always straightforward (see the example at [Appendix B](#)). The reference document on V&V Tools provides a discussion of the types of tools that can be used to perform various V&V techniques.

Common V&V Technique Applications									
Class	V&V Technique	M&S Phase						V&V Category	
		M&S Rqmts	Conceptual Model	M&S Design	M&S Development	M&S Use	M&S Assessment	Verification	Validation
Dyn	Acceptance test ¹					X	X	X	X
Dyn	Alpha test ²				X	X		X	X
Dyn	Assertion check ³				X			X	
Inf	Audit ⁴	X	X	X	X			X	X
Dyn	Authorization test ⁵				X	X	X	X	
Dyn	Beta test ⁶				X	X		X	X
Dyn	Bottom-up test ⁷				X			X	
Dyn	Boundary value test ⁸				X			X	X
Dyn	Branch test ⁹				X			X	
Stat	Calling structure analysis ¹⁰		X	X	X			X	
Stat	Cause-effect graphing ¹¹	X	X	X	X			X	X
Dyn	Comparison test ¹²	X	X	X	X	X			X

Common V&V Technique Applications									
Class	V&V Technique	M&S Phase						V&V Category	
		M&S Rqmts	Conceptual Model	M&S Design	M&S Development	M&S Use	M&S Assessment	Verification	Validation
Stat	Concurrent process analysis ¹³				X	X		X	
Dyn	Condition test ¹⁴				X			X	
Stat	Control flow analysis ¹⁵		X	X	X			X	
Stat	Data dependency analysis ¹⁶		X	X	X			X	
Stat	Data flow analysis ¹⁷		X	X	X			X	
Dyn	Data flow test ¹⁸				X			X	
Dyn	Data interface test ¹⁹				X	X		X	
Dyn	Debugging ²⁰				X			X	X
Inf	Desk check ²¹	X	X	X	X			X	
Inf	Documentation check	X	X		X	X			
Dyn	Equivalence partitioning test ²²				X			X	X
Dyn	Execution monitoring ²³				X	X		X	X
Dyn	Execution profiling ²⁴				X	X		X	X
Dyn	Execution trace ²⁵				X	X		X	X
Dyn	Extreme input test ²⁶				X			X	X
Inf	Face validation ²⁷	X	X	X	X	X	X		X
Stat	Fault/Failure analysis ²⁸				X	X		X	
Dyn	Fault/Failure insertion test ²⁹				X	X		X	X
Dyn	Field test ³⁰					X			X
Dyn	Functional test ³¹				X	X		X	X
Dyn	Graphical comparison ³²				X	X		X	X
For	Induction		X	X					
For	Inductive assertions		X	X				X	
For	Inference		X	X				X	
Inf	Inspection ³³	X	X	X	X	X	X	X	X
Dyn	Invalid input test ³⁴				X	X		X	X
For	Lambda calculus		X	X					
For	Logical deduction		X	X					
Dyn	Loop test ³⁵				X			X	

Common V&V Technique Applications									
Class	V&V Technique	M&S Phase						V&V Category	
		M&S Rqmts	Conceptual Model	M&S Design	M&S Development	M&S Use	M&S Assessment	Verification	Validation
Stat	Model interface analysis ³⁶	X	X	X				X	X
Dyn	Model interface test ³⁷				X	X	X	X	
Dyn	Object-flow test ³⁸				X	X	X	X	
Dyn	Partition test ³⁹			X	X			X	X
Dyn	Path test ⁴⁰				X	X		X	
Dyn	Performance test ⁴¹					X	X	X	
For	Predicate calculus		X	X					
For	Predicate transformation		X	X					
Dyn	Predictive validation ⁴²				X	X	X		X
Dyn	Product test ⁴³					X	X	X	X
For	Proof of Correctness		X	X					
Dyn	Real-time input test ⁴⁴				X	X	X		X
Dyn	Regression test ⁴⁵				X	X		X	
Inf	Review ⁴⁶	X	X	X	X	X	X	X	X
Dyn	Security test ⁴⁷					X	X	X	
Dyn	Self-driven input test ⁴⁸				X	X	X		X
Stat	Semantic analysis ⁴⁹				X	X		X	
Dyn	Sensitivity analysis ⁵⁰				X	X	X	X	X
Dyn	Standards test ⁵¹					X	X	X	
Stat	State transition analysis ⁵²		X	X	X			X	
Dyn	Statement test ⁵³				X	X	X	X	
Dyn	Statistical techniques ⁵⁴				X	X	X		X
Dyn	Stress test ⁵⁵				X	X	X		X
Stat	Structural analysis ⁵⁶		X	X				X	
Dyn	Submodel/Module test ⁵⁷				X				X
Dyn	Symbolic debugging ⁵⁸				X			X	
Stat	Symbolic evaluations ⁵⁹			X				X	
Stat	Syntax analysis ⁶⁰				X			X	
Dyn	Top-down test ⁶¹				X			X	
Stat	Traceability assessment ⁶²		X	X	X	X		X	X

Common V&V Technique Applications									
Class	V&V Technique	M&S Phase						V&V Category	
		M&S Rqmts	Conceptual Model	M&S Design	M&S Development	M&S Use	M&S Assessment	Verification	Validation
Dyn	Trace-driven input test ⁶³				X	X	X		X
Inf	Turing test ⁶⁴				X	X	X		X
Stat	User interface analysis ⁶⁵		X		X	X	X	X	X
Dyn	User interface test ⁶⁶				X	X		X	
Dyn	Visualization/ Animation ⁶⁷				X	X	X	X	X
Inf	Walkthroughs ⁶⁸	X	X	X	X	X	X	X	X

Conducting an effective V&V effort is extremely important for the successful completion of complex and large-scale simulation applications and for resolution of complex problems. How much to test or when to stop testing depends on the requirements of the application or problem involved. The V&V effort should continue until the User obtains sufficient confidence in the credibility and acceptability of the simulation results.

¹ Operationally test the model with actual hardware and data to determine if specified requirements are met.
² Operational testing of initial, complete version of the model at an in-house site uninvolved with the model development.
³ Verification technique – an assertion is a statement that should hold true as the simulation executes and is placed in various parts of the model.
⁴ Assess the application of M&S with respect to established policies, standards.
⁵ Test the implementation in a simulation of security access authorization.
⁶ Developer's operational testing of first release version of complete model at a beta user site.
⁷ Thoroughly test simulation's routines starting from the base level to the highest level.
⁸ Examine accuracy by using test cases on the boundaries of input data classes.
⁹ Use test data to execute as many branch alternatives as possible.
¹⁰ Assess model accuracy by identifying who calls whom, who is called by whom.
¹¹ Identify cause/effects of modeled system, create decision table and convert into test cases with which the model is tested.
¹² Compare results from models of the same system.
¹³ Assess model accuracy by investigating possible synchronization and time management problems in parallel or distributed simulations.
¹⁴ Run under test data to execute as many logical conditions as possible.
¹⁵ Graph the model to examine sequences of control transfer to identify incorrect or inefficient constructs within the model representation.
¹⁶ Determines which variables depend on other variables. Critical for assessing synchronization accuracy across multiple processors.
¹⁷ Assess use of model variables with respect to when variable space is allocated, accessed, and deallocated.
¹⁸ Explore sequences of events related to the status of data structures and examine data flow anomalies.
¹⁹ Ensure that data entering the model and derived from the model are accurately read or stored.

-
- 20 Iterative process to uncover and correct errors or misconceptions.
 - 21 Process of intensely examining work to ensure correctness, completeness, clarity.
 - 22 Test the accuracy of the model with a representative value from each input data class.
 - 23 Gather and examine activity- and event-oriented (low-level) information resulting from model execution.
 - 24 Examine high-level information (profiles) about activities and events during model execution.
 - 25 Reveals errors by reviewing line-by-line execution of a simulation.
 - 26 Assess model at minimum or maximum values for the model inputs.
 - 27 Subjective comparison of model and system behaviors; preliminary approach to validation in the early stages of development.
 - 28 Examine model design specification to determine if any failures logically could occur, and under what conditions.
 - 29 Insert a fault or failure and observe if the model produces the expected invalid behavior.
 - 30 Places the model in an operation situation to collect information for validation.
 - 31 Assesses the accuracy of model input-output transformation.
 - 32 Compare graphs of model variables over time to system variables.
 - 33 Formalized five-step process, checklist approach for uncovering errors.
 - 34 Examine model performance using incorrect input data.
 - 35 Run model to execute as many loop structures as possible.
 - 36 Determines if interface structure and behavior are accurate.
 - 37 Assess how well submodels are integrated with each other.
 - 38 Examine the way a dynamic object flows through activities/processes during its lifetime in model execution.
 - 39 Analyze the model's functional partitions by comparing partitions of the model specification and implementation and testing model with test data.
 - 40 Run model to execute as many control flow paths as possible.
 - 41 Test whether all performance characteristics are measured and evaluated with sufficient accuracy.
 - 42 Use past input data, then compare model outputs with past output data.
 - 43 Preparation for acceptance testing.
 - 44 For simulations representing embedded real-time systems, assess model accuracy using real-time input data.
 - 45 Investigates variable relationships, ensures that modifications do not create other errors.
 - 46 Evaluation relative to specifications and standards by management level team.
 - 47 Assess model using input data sampled from probabilistic models representing random input conditions for a real system.
 - 48 Assess model using input data sampled from probabilistic models representing random input conditions for a real system.
 - 49 The content of the source code as described by the compiler is examined by the modeler to verify that the original intent is accurately reflected.
 - 50 Identify variables/parameters to which model behavior is very sensitive.
 - 51 Substantiates that the M&S application is developed to meet required standards, procedures, and guidelines.
 - 52 Using a state transition diagram, assess model accuracy by analyzing conditions under which a state change occurs.
 - 53 Run model to execute as many statements as possible.
 - 54 Model and system outputs are compared using [multivariate statistical techniques](#) to capture correlation.
 - 55 Assess model validity under extreme workload conditions.
 - 56 Examines model structure for anomalies such as unconditional branches, excessive nesting, multiple entry/exit points.
 - 57 Decompose model into submodels. Compare the behavior of each submodel to the behavior of the corresponding subsystem of the actual system.
 - 58 Debugging tool to manipulate model execution while viewing the model at source code level.
 - 59 Technique that shows path correctness for all computations.
 - 60 Done by the compiler to ensure that language mechanics are correctly applied.
 - 61 Test model at starting at highest level to base level.
 - 62 Used to match elements of one form of the model to another such as requirements specification elements to model design specification.
 - 63 Refine raw trace data collected from a real system for testing a model.

- ⁶⁴ Examination by experts on output data, one from the model and one from the system for feedback in correcting model representation.
- ⁶⁵ Examines user-model interface to prevent errors during user's interactions with model.
- ⁶⁶ Tests human-in-the-loop and interactive simulations.
- ⁶⁷ Display graphical images of model's dynamic behavior during execution.
- ⁶⁸ Meeting to detect and document faults, less formal than inspections.

References

- Ackerman, A.F., Fowler, P.J., & Ebenau, R.G., "Software inspections and the industrial production of software," in Hans-Ludwig Hausen (Ed.), *Software validation: Inspection, testing, verification, alternatives*, Proceedings of the Symposium on Software Validation, pp. 13–40, Darmstadt, FRG, 1983.
- Adrion, W.R., Branstad, M.A., & Cherniavsky, J.C., "Validation, verification, and testing of computer software," *Computing Surveys*, 14 (2), pp. 159–192, 1982.
- Aigner, D.J., "A note on verification of computer simulation models," *Management Science*, 18 (11), pp. 615–619, 1972.
- Allen, F.E. & Cocke, J., "A program data flow analysis procedure," *Communications of the ACM*, 19 (3), pp. 137–147, 1976.
- Backhouse, R.C., *Program construction and verification*, Prentice-Hall International (UK) Ltd., London, 1986.
- Balci, O., "The implementation of four conceptual frameworks for simulation modeling in high-level languages," in M.A. Abrams, P.L. Haigh, & J.C. Comfort (Eds.), *Proceedings of the 1988 Winter Simulation Conference*, pp. 287–295, IEEE, Piscataway, NJ, 1988.
- Balci, O., Bertelrud, A.I., Esterbrook, C.M., & Nance, R.E., "A picture-based object-oriented visual simulation environment," in C. Alexopoulos, K. Kang, W.R. Lilegdon, & D. Goldsman (Eds.), *Proceedings of the 1995 Winter Simulation Conference*, pp. 1333–1340, IEEE, Piscataway, NJ, 1995.
- Balci, O. & Sargent, R.G., "A methodology for cost-risk analysis in the statistical validation of simulation models," *Communications of the ACM*, 24 (4), pp. 190–197, 1981.
- Balci, O. & Sargent, R.G., "Some examples of simulation model validation using hypothesis testing," in H.J. Highland, Y.W. Chao, & O.S. Madrigal (Eds.), *Proceedings of the 1982 Winter Simulation Conference* (pp. 620–620), IEEE, Piscataway, NJ, 1982a.
- Balci, O. & Sargent, R.G., "Validation of multivariate response models using Hotelling's two-sample T^2 test," *Simulation*, 39 (6), pp.185–192, 1982b.
- Balci, O. & Sargent, R.G., "Validation of multivariate response trace-driven simulation models," in A.K. Agrawala & S.K. Tripathi (Eds.), *Performance '83* (pp. 309–323), North-Holland, Amsterdam, 1983.

- Balci, O. & Sargent, R.G., "Validation of simulation models via simultaneous confidence intervals," *American Journal of Mathematical and Management Sciences*, 4 (3&4), pp. 375–406, 1984.
- Banks, J., Carson, J.S., & Nelson, B.L., *Discrete-event system simulation* (2nd ed.), Prentice-Hall, Englewood Cliffs, NJ, 1996.
- Barendregt, H.P., *The lambda calculus: Its syntax and semantics*, North-Holland, New York, 1981.
- Beizer, B., *Software testing techniques* (2nd ed.), Van Nostrand Reinhold, New York, 1990.
- Birta, L.G. & Ozmizrak, F.N., "A knowledge-based approach for the validation of simulation models: The foundation," *ACM Transactions on Modeling and Computer Simulation* (in press), 1996.
- Chusho, T., "Test data selection and quality estimation based on the concept of essential branches for path testing," *IEEE Transactions on Software Engineering*, SE-13 (5), pp. 509–517, 1987.
- Cohen, K.J. & Cyert, R.M., "Computer models in dynamic economics," *Quarterly Journal of Economics*, 75 (1), pp. 112–127, 1961.
- Damborg, M.J. & Fuller, L.F., "Model validation using time and frequency domain error measures," (ERDA Report No. 76-152), NTIS, Springfield, VA, 1976.
- Deutsch, M.S., *Software verification and validation: Realistic project approaches*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
- Dijkstra, E.W., "Guarded commands, non-determinacy and a calculus for the derivation of programs," *Communications of the ACM*, 18 (8), pp. 453–457, 1975.
- Dillon, L.K. (1990). Using symbolic execution for verification of Ada tasking programs. *ACM Transactions on Programming Languages and Systems*, 12 (4), 643–669.
- Dobbins, J.H., "Inspections as an up-front quality technique," in G.G. Schulmeyer & J.I. McManus (Eds.), *Handbook of Software Quality Assurance*, pp. 137–177, Van Nostrand-Reinhold Company, New York, 1987.
- Dunn, R.H., *Software defect removal*, McGraw-Hill, New York, 1984.
- Dunn, R.H., "The quest for software reliability," in G.G. Schulmeyer & J.I. McManus (Eds.), *Handbook of Software Quality Assurance*, pp. 342–384, Van Nostrand-Reinhold Company, New York, 1987.
- Emshoff, J.R. & Sisson, R.L., *Design and use of computer simulation models*, MacMillan, New York, 1970.
- Fairley, R.E., "An experimental program-testing facility," *IEEE Transactions on Software Engineering*, SE-1 (4), pp. 350–357, 1975.
- Fairley, R.E., "Dynamic testing of simulation software," in *Proceedings of the 1976 Summer Computer Simulation Conference*, pp. 708–710, Simulation Councils, La Jolla, CA, July, 1976.

- Fishman, G.S. & Kiviat, P.J., "The analysis of simulation generated time series," *Management Science*, 13 (7), pp. 525–557, 1987.
- Forrester, J.W., *Industrial Dynamics*, MIT Press, Cambridge, MA, 1961.
- Fujimoto, R.M., "Parallel discrete event simulation," *Communications of the ACM*, 33 (10), pp. 31–53, 1990.
- Fujimoto, R.M., "Parallel discrete event simulation: Will the field survive?," *ORSA Journal on Computing*, 5 (3), pp. 213–230, 1993.
- Gafarian, A.V. & Walsh, J.E., "Statistical approach for validating simulation models by comparison with operational systems," in *Proceedings of the 4th International Conference on Operations Research*, pp. 702–705, John Wiley & Sons, New York, 1969.
- Gallant, A.R., Gerig, T.M., & Evans, J.W., "Time series realizations obtained according to an experimental design," *Journal of the American Statistical Association*, 69 (347), pp. 639–645, 1974.
- Garratt, M., "Statistical validation of simulation models," in *Proceedings of the 1974 Summer Computer Simulation Conference*, pp. 915–926, Simulation Councils, La Jolla, CA, July, 1974.
- Hermann, C.F., "Validation problems in games and simulations with special reference to models of international politics," *Behavioral Science*, 12 (3), pp. 216–231, 1967.
- Hollocker, C.P., "The standardization of software reviews and audits," in G.G. Schulmeyer & J.I. McManus (Eds.), *Handbook of Software Quality Assurance*, pp. 211–266, Van Nostrand-Reinhold Company, NY, 1987.
- Howden, W.E., "Reliability of the path analysis testing strategy," *IEEE Transactions on Software Engineering*, SE-2 (3), pp. 208–214, 1976.
- Howden, W.E., "Functional program testing," *IEEE Transactions on Software Engineering*, SE-6 (2), pp. 162–169, 1980.
- Howrey, P. & Kelejian, H.H., "Simulation versus analytical solutions," in T.H. Naylor (Ed.), *The design of computer simulation experiments*, pp. 207–231, Duke University Press, Durham, NC, 1969.
- Hunt, A.W., "Statistical evaluation and verification of digital simulation models through spectral analysis," unpublished doctoral dissertation, University of Texas at Austin, 1970.
- Khanna, S., "Logic programming for software verification and testing," *The Computer Journal*, 34 (4), pp. 350–357, 1991.
- Kheir, N.A. & Holmes, W.M., "On validating simulation models of missile systems," *Simulation*, 30 (4), pp. 117–128, 1978.
- King, J.C., "Symbolic execution and program testing," *Communications of the ACM*, 19 (7), 385–394, 1976.
- Kleijnen, J.P.C., *Statistical Techniques in Simulation* (Vol. 2), Marcel Dekker, NY, 1975.

- Knight, J.C. & Myers, E.A., "An improved inspection technique," *Communications of the ACM*, 36 (11), pp. 51–61, 1993.
- Law, A.M. & Kelton, W.D., *Simulation modeling and analysis* (2nd ed.), McGraw-Hill, NY, 1991.
- Manna, Z., Ness, S., & Vuillemin, J., "Inductive methods for proving properties of programs," *Communications of the ACM*, 16 (8), pp. 491–502, 1973.
- Miller, D.K., "Validation of computer simulations in the social sciences," in *Proceedings of the Sixth Annual Conference on Modeling and Simulation*, (pp. 743–746). Pittsburg, PA., 1975.
- Miller, D.R., "Model validation through sensitivity analysis," in *Proceedings of the 1974 Summer Computer Simulation Conference*, pp. 911–914, Simulation Councils, La Jolla, CA, July, 1974a.
- Miller, D.R., "Sensitivity analysis and validation of simulation models," *Journal of Theoretical Biology*, 48 (2), pp. 345–360, 1974b.
- Miller, L.A., Groundwater, E.H., Hayes, J.E., & Mirsky, S.M., "Survey and assessment of conventional software verification and validation methods," *Special Publication NUREG/CR-6316, Vol. 2*, U.S. Nuclear Regulatory Commission, Washington, DC, 1995.
- Myers, G.J., "A controlled experiment in program testing and code walkthroughs/inspections," *Communications of the ACM*, 21 (9), pp. 760–768, 1978.
- Myers, G.J., *The art of software testing*, John Wiley & Sons, NY, 1979.
- Naylor, T.H. & Finger, J.M. "Verification of computer simulation models," *Management Science*, 14 (2), B92–B101, 1967.
- Ould, M.A. & Unwin, C., *Testing in software development*, Cambridge University Press, Great Britain, 1986.
- Page, E.H. & Nance, R.E., "Parallel discrete event simulation: A modeling methodological perspective," in D.K. Arvind, R. Bagrodia, & J.Y-B. Lin (Eds.), *Proceedings of the Eighth Workshop in Parallel and Distributed Simulation (PADS '94)*, pp. 88–93, IEEE Computer Society Press., Los Alamitos, CA, July, 1994.
- Paul, R.J., "Visual simulation: Seeing is believing?," in R. Sharda, B.L. Golden, E. Wasil, O. Balci, & W. Stewart (Eds.), *Impacts of recent computer advances on operations research*, pp. 422–432, Elsevier, NY, 1989.
- Perry, W., *Effective methods for software testing*, John Wiley & Sons, NY, 1995.
- Prather, R.E. & Myers, J.P., Jr, "The path prefix software testing strategy," *IEEE Transactions on Software Engineering*, SE-13 (7), pp. 761–766, 1987.
- Pressman, R.S., *Software engineering: A practitioner's approach* (4th Ed.), McGraw-Hill, NY, 1996.

- Ramamoorthy, C.V., Ho, S.F., & Chen, W.T., "On the automated generation of program test data," *IEEE Transactions on Software Engineering*, SE-2 (4), pp. 293–300, 1976.
- Rattray, C. (Ed.), *Specification and verification of concurrent systems*, Springer-Verlag, NY, 1990.
- Reynolds, C. & Yeh, R.T. "Induction as the basis for program verification," *IEEE Transactions on Software Engineering*, SE-2 (4), pp. 244–252, 1976.
- Richardson, D.J. & Clarke, L.A., "Partition analysis: A method combining testing and verification," *IEEE Transactions on Software Engineering*, SE-11 (12), pp. 1477–1490, 1985.
- Rowland, J.R. & Holmes, W.M., "Simulation validation with sparse random data," *Computers and Electrical Engineering*, 5 (3), pp. 37–49, 1978.
- Sargent, R.G., "Validation and verification of simulation models," in J.J. Swain, D. Goldsman, R.C. Crain, & J.R. Wilson (Eds.), *Proceedings of the 1992 Winter Simulation Conference*, pp. 104–114, IEEE, Piscataway, NJ, 1992.
- Schach, S.R., "Software engineering (3rd ed.), Irwin, Homewood, IL, 1996.
- Schruben, L.W. "Establishing the credibility of simulations," *Simulation*, 34 (3), pp. 101–105, 1980.
- Shannon, R.E., *Systems simulation: The art and science*, Prentice-Hall, Englewood Cliffs, NJ, 1975.
- Sommerville, I., *Software engineering* (5th ed.), Addison-Wesley, Reading, MA, 1996.
- Teorey, T.J., Validation criteria for computer system simulations, *Simuletter*, 6 (4), pp. 9–20, 1975.
- Theil, H., *Economic forecasts and policy*, North-Holland, Amsterdam, The Netherlands, 1961.
- Turing, A.M., "Computing machinery and intelligence," in E.A. Feigenbaum & J. Feldman (Eds.), *Computers and thought*, pp. 11–15, McGraw-Hill, NY, 1963.
- Tytula, T.P., *A method for validating missile system simulation models*, (Technical Report E-78-11), U.S. Army Missile R&D Command, Redstone Arsenal, AL, June, 1978.
- Van Horn, R.L., "Validation of simulation results," *Management Science*, 17 (5), pp. 247–258, 1971.
- Watts, D, Time series analysis," in T.H. Taylor (Ed.), *The design of computer simulation experiments*, pp. 165–179, Duke University Press, Durham, NC, 1969.
- Whitner, R.B. & Balci, O., "Guidelines for selecting and using simulation model verification techniques," in E.A. MacNair, K.J. Musselman, & P. Heidelberger (Eds.), *Proceedings of the 1989 Winter Simulation Conference*, pp. 559–568, IEEE, Piscataway, NJ, 1989.

- Wright, R.D., "Validating dynamic models: An evaluation of tests of predictive power," in *Proceedings of the 1972 Summer Computer Simulation Conference* (pp. 1286–1296), Simulation Councils, La Jolla, CA, 1972.
- Yourdon, E., *Structured Walkthroughs* (3rd ed.), Yourdon Press, NY, 1985.
- Yucesan, E. & Jacobson, S.H. (1992). Building correct simulation models is difficult. In J.J. Swain, D. Goldsman, R.C. Crain, & J.R. Wilson (Eds.), *Proceedings of the 1992 Winter Simulation Conference*, pp. 783–790, IEEE, Piscataway, NJ, 1992.
- Yucesan, E. & Jacobson, S.H., "Intractable structural issues in discrete event simulation: Special cases and heuristic approaches," *ACM Transactions on Modeling and Computer Simulation* (in press), 1996.
- Yeh, R.T., "Verification of programs by predicate transformation," in *Current Trends in Programming Methodology*, Vol. 2 (pp. 228–247). Prentice-Hall, Englewood Cliffs, 1977.

RPG References in this Document

- select menu: *RPG Core Documents*, select item: "V&V Agent Role in the VV&A of Federations"
- select menu: *RPG Core Documents*, select item: "V&V Agent Role in the VV&A of Legacy Simulations"
- select menu: *RPG Core Documents*, select item: "V&V Agent Role in the VV&A of New Simulations"
- select menu: *RPG Diagrams*, select item: "Overall Problem Solving Process"
- select menu: *RPG Diagrams*, select item: "VV&A and Federation Construction"
- select menu: *RPG Diagrams*, select item: "VV&A and Legacy Simulation Preparation"
- select menu: *RPG Diagrams*, select item: "VV&A and New M&S Development"
- select menu: *RPG Reference Documents*, select item: "V&V Tools"
- select menu: *RPG Special Topics*, select item: "Subject Matter Experts and VV&A"

Additional References

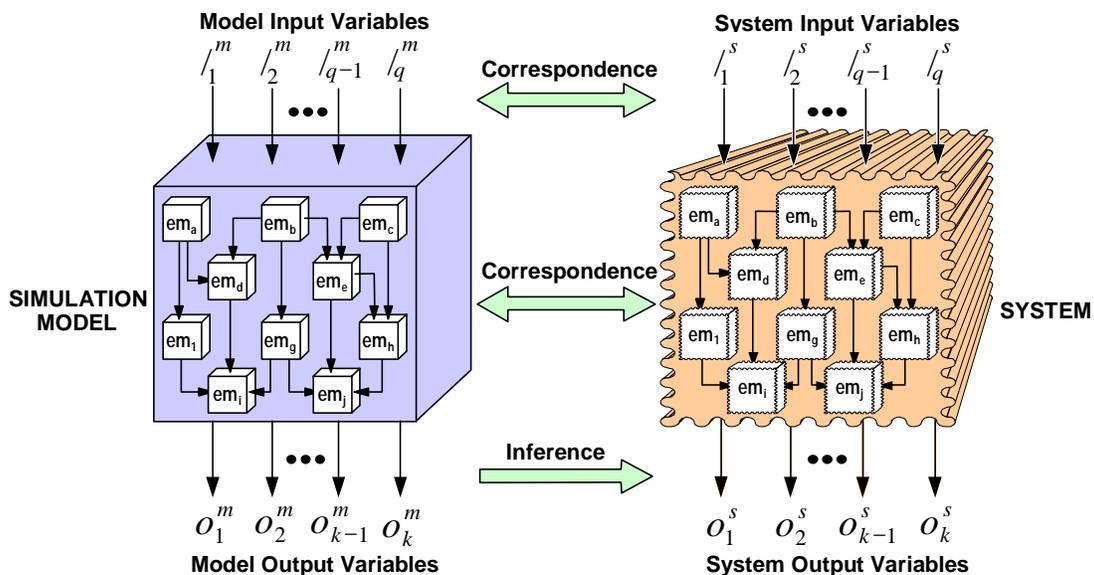
- Balci, O., "Requirements for model development environment," *Computers & Operations Research*, 13 (1), pp. 53–67, 1986.
- Balci, O. & Nance, R.E., "Simulation model development environments: A research prototype," *Journal of Operational Research Society*, 38 (8), pp. 753–763, 1987.
- Derrick, E.J. & Balci, O., "A visual simulation support environment based on the DOMINO conceptual framework," *Journal of Systems and Software*, 31 (3), pp. 215–237, 1995.

- Moose, R.L. & Nance, R.E., "The design and development of an analyzer for discrete event model specifications," in R. Sharda, B.L. Golden, E. Wasil, O. Balci, & W. Stewart (Eds.), *Impacts of recent computer advances on operations research*, pp. 407–421, Elsevier, NY, 1989.
- Nance, R.E. & Overstreet, C.M., "Diagnostic assistance using digraph representations of discrete event simulation model specifications," *Transactions of the SCS*, 4 (1), pp. 33–57, 1987.
- Overstreet, C.M. & Nance, R.E., "A specification language to assist in analysis of discrete event simulation models," *Communications of the ACM*, 28 (2), pp. 190–201, 1985.
- Stucki, L.G., "New directions in automated tools for improving software quality," in R. Yeh (Ed.), *Current trends in programming methodology*, Vol. 2 (pp. 80–111, Prentice-Hall , Englewood Cliffs, NJ 1977.

Appendix A: Validation Procedure Using Simultaneous Confidence Intervals

The behavioral accuracy (validity) of a simulation with multiple outputs can be expressed in terms of the differences between the corresponding model and system output variables when the model is run with the same input data and operational conditions that drive the real system. The range of accuracy of the *j*th model output variable can be represented by the *j*th confidence interval (c.i.) for the differences between the means of the *j*th model and system output variables. The simultaneous confidence intervals (s.c.i.) formed by these confidence intervals are called the model range of accuracy (m.r.a.) [Balci and Sargent, 1984].

Assume that there are *k* output variables from the model and *k* output variables from the system as shown in the figure below.



Model and System Characteristics

Let $(\underline{\mu}^m)' = [\mu_1^m, \mu_2^m, \dots, \mu_k^m]$ and $(\underline{\mu}^s)' = [\mu_1^s, \mu_2^s, \dots, \mu_k^s]$ be the *k* dimensional vectors of the population means of the model and system output variables, respectively.

Basically, there are three approaches for constructing the s.c.i. to express the m.r.a. for the mean behavior.

In Approach 1, the m.r.a. is determined by the $100(1-\gamma)\%$ s.c.i. for $\underline{\mu}^m - \underline{\mu}^s$ as

$$(1) \quad [\underline{\delta} - \underline{\tau}]$$

where $\underline{\delta}' = [\delta_1, \delta_2, \dots, \delta_k]$ represents lower bounds and $\underline{\tau}' = [\tau_1, \tau_2, \dots, \tau_k]$ represents upper bounds of the s.c.i. The modeler can be $100(1-\gamma)\%$ confident that the true differences between the population means of the model and system output variables are simultaneously contained within (1).

In Approach 2, the $100(1-\gamma^m)\%$ s.c.i. are first constructed for $\underline{\mu}^m$ as

$$(2) \quad [\underline{\delta}^m, \underline{\tau}^m]$$

where $(\underline{\delta}^m)' = [\delta_1^m, \delta_2^m, \dots, \delta_k^m]$ and $(\underline{\tau}^m)' = [\tau_1^m, \tau_2^m, \dots, \tau_k^m]$. Then, the $100(1-\gamma^s)\%$ s.c.i. are constructed for $\underline{\mu}^s$ as

$$(3) \quad [\underline{\delta}^s, \underline{\tau}^s]$$

where $(\underline{\delta}^s)' = [\delta_1^s, \delta_2^s, \dots, \delta_k^s]$ and $(\underline{\tau}^s)' = [\tau_1^s, \tau_2^s, \dots, \tau_k^s]$. Finally, using the Bonferroni inequality, the m.r.a. is determined by the following s.c.i. for $\underline{\mu}^m - \underline{\mu}^s$ with a confidence level of at least $(1-\gamma^m - \gamma^s)$ when the model and system outputs are dependent and with a level of at least $(1-\gamma^m - \gamma^s + \gamma^m\gamma^s)$ when the outputs are independent [Kleijnen, 1975]:

$$(4) \quad [\underline{\delta}^m - \underline{\tau}^s, \underline{\tau}^m - \underline{\delta}^s]$$

In Approach 3, the model and system output variables are observed in pairs and the m.r.a. is determined by the $100(1-\gamma)\%$ s.c.i. for $\underline{\mu}^d$, the population means of the differences of paired observations, as

$$(5) \quad [\underline{\delta}^d, \underline{\tau}^d]$$

where $(\underline{\delta}^d)' = [\delta_1^d, \delta_2^d, \dots, \delta_k^d]$ and $(\underline{\tau}^d)' = [\tau_1^d, \tau_2^d, \dots, \tau_k^d]$.

The m.r.a. is constructed with the observations derived from the model and system output variables by running the model with the same input data and operational conditions that drive the real system. If the simulation is self-driven, then the model

input data come independently from the same populations or stochastic process as the system input data. Because the model and system input data are independent of each other, but come from the same populations, the model and system output data are expected to be independent and identically distributed. Hence, Approach 1 or 2 can be used. The use of Approach 3 in this case would be less efficient. If the simulation is trace-driven, the model input data are exactly the same as the system input data. In this case, the model and system output data are expected to be dependent and identical. Therefore, Approach 2 or 3 should be used.

Sometimes, the model or simulation application sponsor or proponent may specify an acceptable range of accuracy for a specific simulation. This specification can be made for the mean behavior of a stochastic simulation as

$$(6) \quad \underline{L} \leq \underline{\mu}^m - \underline{\mu}^s \leq \underline{U}$$

where $\underline{L}' = [L_1, L_2, \dots, L_k]$ and $\underline{U}' = [U_1, U_2, \dots, U_k]$ are the lower and upper bounds of the acceptable differences between the population means of the model and system output variables. In this case, the m.r.a. should be compared against Equation (6) to evaluate model validity.

The shorter the lengths of the m.r.a., the more meaningful is the information they provide. The lengths can be decreased by increasing the sample sizes or by decreasing the confidence level. Such increases in sample sizes, however, may increase the cost of data collection. Thus, a trade-off analysis may be necessary among the sample sizes, confidence levels, half-length estimates of the m.r.a., data collection method, and cost of data collection. For details of performing the trade-off analysis, see Balci and Sargent, 1984.

Appendix B: Selecting V&V Techniques for Defect Detection

The table below shows the subset of the V&V techniques listed in the Common V&V Technique Applications table that are considered useful for detecting software defects. This subset was analyzed by a survey done to assess the types and frequency of defects detected using conventional software verification methods [SAIC, 1993]. In the survey, 52 types of defects were assessed, the defects falling into one of three categories: requirements, design, and code.

Capability of V&V Techniques to Detect Software Defects			
V&V Technique	Requirements Defect Types Detected (Out of 13)	Design Defect Types Detected (Out of 15)	Code Defect Types Detected (Out of 24)
Assertion check	0	0	2
Branch test	0	2	6
Calling structure analysis	1	5	15
Cause-effect graphing	2	2	1
Concurrent process analysis	0	0	9
Condition test	0	0	12
Control flow analysis	1	3	9
Data flow analysis	3	0	12
Data interface test	2	4	3
Desk checking	0	0	5
Fault/Failure insertion test	0	0	19
Field test	0	15	24
Inductive assertions	4	4	14
Inspections			
Path test		2	8
Regression test	0	0	24
State transition analysis	1	3	9
Statement test	0	0	2
Stress test	0	15	24
User interface analysis	8	8	3
Walkthroughs	0	0	14

The reference acknowledges that the survey was subjective in nature and ignored questions of how well or how easily a defect could be found by a particular method. However, the goal of the survey was to determine what techniques might be expected

to detect a particular type of flaw. In all, 13 requirements defects, 15 design defects, and 24 code defects are defined. Survey results for all three categories are summarized in the table below.

Results indicate that a user interface analysis is expected to detect most of the study's pre-defined 13 requirements defects, while field testing and stress testing are expected to detect all of the 15 identified design defects. Top techniques for detecting the code defects include field testing, stress testing, and regression testing. Similar results are expected when applying these techniques to V&V of M&S.

The appearance of hyperlinks does not constitute endorsement by the DoD, DMSO, the administrators of this web site, or the information, products or services contained therein. For other than authorized activities such as military exchanges and Morale, Welfare and Recreation sites, the DoD does not exercise any editorial control over the information you may find at these locations. Such links are provided consistent with the stated purpose of this DMSO web site.

§ § § § § § §