

**Verification Technology Potential with Different Modeling and Simulation
Development and Implementation Paradigms**

Garth R. MacKenzie

G. Gordon Schulmeyer

Levent Yilmaz

An Invited Paper for Session A1

Foundations for V&V in the 21st Century Workshop (Foundations '02)

The Johns Hopkins University Applied Physics Laboratory

Laurel, Maryland (USA)

October 22-24, 2002

Verification Technology Potential with Different Modeling and Simulation Development and Implementation Paradigms

Garth R. MacKenzie

University of Maryland
University College
3501 University Blvd. East
Adelphi, MD 20783
(301) 985-7092
gmackenzie@umuc.edu

G. Gordon Schulmeyer

PYXIS Systems International,
Incorporated
1120C Benfield Blvd.
Suite 121
Millersville, MD 21108
(410) 729-0416
pyxis@pyxisinc.com

Levent Yilmaz

Trident Systems, Incorporated
10201 Lee Hwy.
Suite 300
Fairfax, VA 22030
(703) 267-6754
levent@tridsys.com

Abstract

An investigation of the history of verification and validation (V&V) in modeling and simulation (M&S) indicates that validity has been interpreted in different ways based on the context and the types of problems being solved. That is, advances in modeling paradigms, methodologies, tools, and technologies had considerable impact on the way validation is perceived. The shift from an accuracy point of view to a quality-centric approach in model V&V is a recent indicator of this trend. The modeling artifacts produced by a certain paradigm usually depict specific characteristics unique to that paradigm such that widely adopted conventional methods do not properly scale or apply. This paper introduces a generic simulation lifecycle and discusses the V&V processes in the context of the life cycle of a simulation study. Key standards, practices, and processes for V&V of software-intensive models are reviewed. Special emphasis is given to major V&V issues as well as emergent challenges due to a variety of modern simulation implementation paradigms and development methodologies. The issues include recognizing and avoiding some of the myths of simulation model V&V to focus on essential challenges as opposed to accidentally perceived problems. Building on the identified issues and challenges with regard to the emerging prominent trends in model development, a set of potential research directions are identified and discussed.

1. Introduction

Simulation modeling is a process that entails the development of a model of a system to conduct experiments for the purpose of understanding the behavior of the system and evaluating its alternative operation strategies. As the complexity of models increase, technologies evolve, and development paradigms change, the application of traditional analytical methods are less likely to scale to facilitate reasoning about emergent behaviors of large-scale complex systems. Hence, simulation modeling is becoming one of the most powerful problem solving methods for scientists and engineers to reason about the processes of complex systems that they develop and examine. However, unless developed models are demonstrated to be valid and certified to be credible, the predictions and explanations resulting from simulation experiments will carry the high risk of leading to the dissemination of inaccurate knowledge and decisions. Model validation and verification play key roles in mitigating such risks.

Model validation is defined as demonstrating that a computerized model satisfies the simulation objectives and requirements with sufficient accuracy within its domain of applicability. A model is considered to be valid under a set of experimental conditions if the model's response accuracy is within acceptable range for its intended purpose [Sargent 1996]. The model validation perspectives in operations research heavily influenced this traditional view of simulation model validity. A close look at the model validation history indicates that validity has been interpreted in different ways based on the context and the types of problems being solved. That is, advances in modeling paradigms, methodologies, tools, and technologies had considerable impact on the way validation is perceived. These advances are partly influenced by the changing nature of the problems attacked by the modeling community. This paper, for instance, examines the validation and verification technologies for software-intensive simulation models under different development and implementation paradigms.

1.1 Definitions

Deutsch [1982] and Boehm [1984] describe software validation as a set of activities designed to guarantee that the right product is being built (from a user's perspective) and verification as activities that guarantee the product is being built correctly - according to requirements specifications, design documentation, and process standards.

Lewis [1992] discusses the value and need for Independent Verification and Validation (IV&V). He establishes six rules that must be observed if the process is to be effective [p. 13]:

- ❑ IV&V must be an independent third-party activity.
- ❑ IV&V must be an overlay, not an integral part of the development cycle; it is an added-value concept.
- ❑ IV&V must report to and owes its fundamental allegiance to the customer, not the developing contractor or organization.
- ❑ Although it may share some of the tools and data used by the development organization, IV&V must also provide its own tools and disciplines
- ❑ Although flexible in terms of its starting point, IV&V must verify each phase relative to itself and to its adjacent phases.
- ❑ IV&V must provide a means of validating all testable software performance requirements.

Lewis [1992] and others [Arthur and Nance 2000; Schulmeyer and MacKenzie 2000; Rakitin 2001] review the cost versus benefits issue associated with IV&V. In addition to establishing a completely independent V&V organization, other organizational alternatives include assigning V&V responsibilities to another group (e.g. Software Quality Assurance) which is separate from the development project team but is still a part of the development contractor's overall organization.

Another issue involves the transferability of various V&V methods and standards to modern, and often less formal, software development paradigms. (A brief list of existing V&V standards and publications is given in Table 1). Finally there is the issue of interpreting and applying V&V in the context of the modeling and simulation application domain. These issues will be discussed in subsequent sections of this paper.

1.2 Verification and Validation for Modeling and Simulation

To better understand the emergent validation approaches and their rationale, it is useful to be aware of the historical evolution of model validation philosophies. For the purpose of our paper, we use the classification advocated in Landry *et al.* [1983] and Derry *et al.* [1993]. In this classification the traditional reductionist/logical positivist school would see a valid model as an objective representation of the system under study. That is, the model is either correct or incorrect for the domain of application. On the other hand, pragmatist and holistic schools that promote systems thinking viewpoint would consider a model valid on the basis of qualitative and subjective evaluations of its contextual usefulness. In this school of thought, a model is not considered to be absolutely correct or incorrect, but rather measurement of qualitative characteristics are considered essential for acceptability and usability. Since simulation model validity means “adequacy with respect to a purpose”, validation needs to have qualitative and subjective evaluation components. The detailed discussion of the above philosophical influences is beyond the scope of this paper; therefore, we refer the reader to Naylor and Finger [1968] and Derry *et al.* [1993], which provide a detailed overview of major schools of thought in philosophy of science that affected validation during the early years of simulation modeling.

As the complexity and nature of today’s simulation application domains, tools, and technologies evolve from small-scale software programs to complex, distributed, concurrent federations of software-intensive models that are integrated with virtual and real-time systems, the simulation development process is increasingly being dominated by software-intensive system development activities. The essential difficulties of software such as irreducible complexity, conformity, and maintainability [Brooks 1995] impacts the acceptability, usability, as well as credibility of models. As a result, the original accuracy-centered validity viewpoint of confidence building needs to be enriched by quality-centric viewpoint by using relevant software verification, validation, and quality assurance activities.

To this end, the simulation modeling community incorporates the notion of model specification, design, and implementation verification stages to the traditional validation process framework of operations researchers. Sargent [1996] defines model verification as the process of ensuring that the programmed model (i.e., software implementation) is a correct realization of the conceptual model of a simulation. A more extensive view of model verification is defined as the activity of demonstrating that the artifacts produced at the end of a simulation development life cycle stage conforms to the requirements set forth at the beginning of this phase. In this framework, model verification deals with the transformational accuracy in model development and works by demonstrating that model development artifacts are transformed from one form to another with sufficient accuracy throughout the development life cycle [Balci 1994]. In other words, as suggested in [Boehm 1984], while validation deals with the issue of “building the right model”, verification is concerned with “building the model right”.

Table 1. Sample V&V Standards List

AFSC/AFLCP 800-5 - *Software Independent Verification and Validation*.

Air Force Instruction 16-1001, *Verification, Validation And Accreditation (VV&A)*, 1 June 1996.

ANS 10.4 - *Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry*, 1987.

DI-M-2051A *Technical Manual Quality Assurance Data*, Navy, 8 Sept. 1976.

DI-MCCR-80770, *Software Independent Verification and Validation Plan*, 21 Feb. 1989.

FHWA Handbook, Wentworth, James A., Knaus, Rodger, and Aougab, Hamid, *Verification, Validation, and Evaluation of Expert Systems: An FHWA Handbook*, Version 1.2 - 1st Edition (McLean: Federal Highway Administration, Jan. 1997).

FIPSPUB 101 - *Guideline for Life cycle Validation, Verification, and Testing of Computer Software*.

FIPSPUB 132 - *Guideline for Software Verification and Validation Plans*, 19 Nov. 1987.

IEEE 1012 *Standard for Software Verification and Validation Plans*, 1988.

IEEE Standard 1074, *Software Quality Management Process and Verification and Validation Process*, 30 Oct. 1992.

IEEE/EIA 12207.0-1996 - *Software Life Cycle Processes*, March 1998.

IEEE/EIA P12207.1 (Draft) *Guide for information technology: - Software life cycle processes, - Life cycle data*, 11 February 1997.

IEEE/EIA 12207.2-1997 *Software life cycle processes - Implementation considerations*, April 1998.

ISO 9001, *Quality Systems - Model for Quality Assurance in Design, Development, Production, Installation, and Servicing*, (Geneva, International Organization for Standardization, 1994).

ISO 9000-3, *Quality management and quality assurance standards - Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software*, (Geneva, International Organization for Standardization, 1991).

JPL D 576 - *Independent Verification and Validation of Computer Software Methodology*, 1983.

MIL-M-81203, *Validation / Verification Plan*.

MIL-STD-499B, *Systems Engineering Management Plan (SEMP)*.

NASA-GB-002-95, *Formal Methods Specification and Verification Guidebook for Software and Computer Systems*, Volume I: Planning and Technology Insertion, Release 1.0, kemp@ivv.nasa.gov.NBS.

NBS Special Publication 500-93, *Software Validation, Verification, and Testing Technique and Tool Reference Guide*, National Bureau of Standards, Computer Science and Technology, Washington, DC, Sept., 1982.

NIST Special Publication 500-234, Wallace, Dolores R., Ippolito, Laura M. and Cuthill, Barbara, *Reference Information for the Software Verification and Validation Process*, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, 29 March 1996.

UDI-M-23928 *Validation and Verification Plan*, Navy-SH, 2 Jan. 1973.

[Schulmeyer and MacKenzie (2000), p. 47]

Taking this point of view, [Balci 1994], adapts a taxonomy of software verification, validation, and testing techniques to the simulation modeling domain. Similarly, Defense Modeling and Simulation Office (DMSO) has recently established a comprehensive *VV&A Recommended Practices Guide* along with a policy definition, DOD Instruction 5000.61, to assign responsibilities and prescribe procedures for the VV&A of DOD M&S. The recommended techniques emphasize computerized model V&V methods applicable throughout the development cycle of simulation studies.

1.3 Authors' Background

Garth MacKenzie is an Associate Professor and Program Director in the Graduate School at University of Maryland University College. He has over 30 years of industry experience as a programmer, systems engineer, manager, and consultant. He has taught graduate level courses in software project management, software verification and validation, CASE tools, data communications, and database systems. He holds BA in Physics, BS in Mathematics, and Master of Administrative Science degrees from the Johns Hopkins University. He is a member of the ACM and a Senior Member of the IEEE.

G. Gordon Schulmeyer is President and co-founder of PYXIS Systems International, Incorporated - a consulting firm that specializes in software project management practices and software process improvement. In addition to his more than 40 years of software development and management experience, he has authored and edited many publications in the field, including the *Handbook of Software Quality Assurance*, *Zero Defect Software*, and *Verification and Validation of Modern Software-Intensive Systems* (with G. MacKenzie as co-author). He holds BS in Mathematics and MBA degrees from Loyola College of Baltimore and a JD degree from the University of Baltimore.

Levent Yilmaz is a senior research engineer in the Simulation and Software Division of Trident Systems Incorporated, in Fairfax, Virginia. He is also an Adjunct Assistant Professor in the Graduate School at University of Maryland University College. He has served as a principle investigator and lead project engineer for a variety of model-based verification technology development efforts. His current research focuses on bringing and applying compositional verification technology and formal models of behavioral consistency analysis and testing to parameterized Synthetic Natural Environment development for Distributed Mission Training. He received a BS degree in Computer Engineering from Bilkent University and holds MS and PhD degrees in Computer Science from Virginia Polytechnic Institute and State University.

1.4 Outline of the Paper

This paper is structured as follows: Section 2 introduces the model development life cycle along with its credibility assessment stages. The key V&V processes are also discussed throughout the presented development life cycle. Section 3 focuses on the major V&V issues and challenges involved with the modern model development and implementation paradigms. In section 3, a set of model validation myths are presented to distinguish essential issues from accidental challenges. Section 4 presents a set of potential future research directions in light of the problems

presented under section 3 as well as emerging trends. Sections 5 and 6 provide recommendations and conclusions, respectively.

2. Verification, Validation, and Testing Throughout the Simulation Model Development Life Cycle

Verification and validation is not a separate phase that is independent of the model building process. Model building and V&V are interlinked, and V&V is a continuous activity throughout the entire simulation model development life cycle. A variety of life cycles are presented in the simulation modeling literature [Nance 1987; Sargent 1996]. For the purpose of this paper, we present a simplified process model similar to the problem solving process depicted in Sargent [1996] and Landry *et al.* [1993]. The goal is to present the main life cycle phases and indicate how the verification and validation process is integrated with the model building life cycle.

2.1 Simulation Model Development Life Cycle

The life cycle of simulation model development is depicted in Figure 1. The stages of the life cycle are shown as solid rectangles. The model development as well as validation processes that relate the phases of the life cycle are shown as arrows. The solid arrows denote the model development processes, while the dashed arrows indicate the validation processes.

The problem domain analysis starts with the system entity or problem situation. System analysts create abstractions on the system structure and behavior to derive system theories and properties of interest. If system data exists due to experimentation with the system, the analysts generalize and hypothesize from the results to facilitate the derivation of the formulated problem definition. Several characteristics of interest are identified in [Shannon 1975] to guide and structure the problem formulation process: (1) change, (2) environment, (3) counterintuitive behavior, (4) drift to low performance, (5) interdependency, (6) organization. Balci [1994] briefly summarizes these characteristics. The solution domain entails the formulation of the simulation requirements and objectives, model formulation through analysis and conceptualization, model specification and implementation, experimentation, and revision. Each of these phases is briefly discussed below.

2.1.1 Requirements and Simulation Objectives Formulation

Requirements formulation process takes formulated problem definition as input to generate simulation objectives and requirements definition. The goal is to bridge the gap between system level objectives definition and simulation model domain. The process is usually divided into three areas of effort: (1) problem recognition, (2) evaluation and synthesis, (3) simulation context (domain) modeling. It is important to understand modeling in a system context. This requires reviewing the model scope in such a way that credible problem recognition is ensured. The objective is recognition of the basic problem entities as perceived by the customer/sponsor.

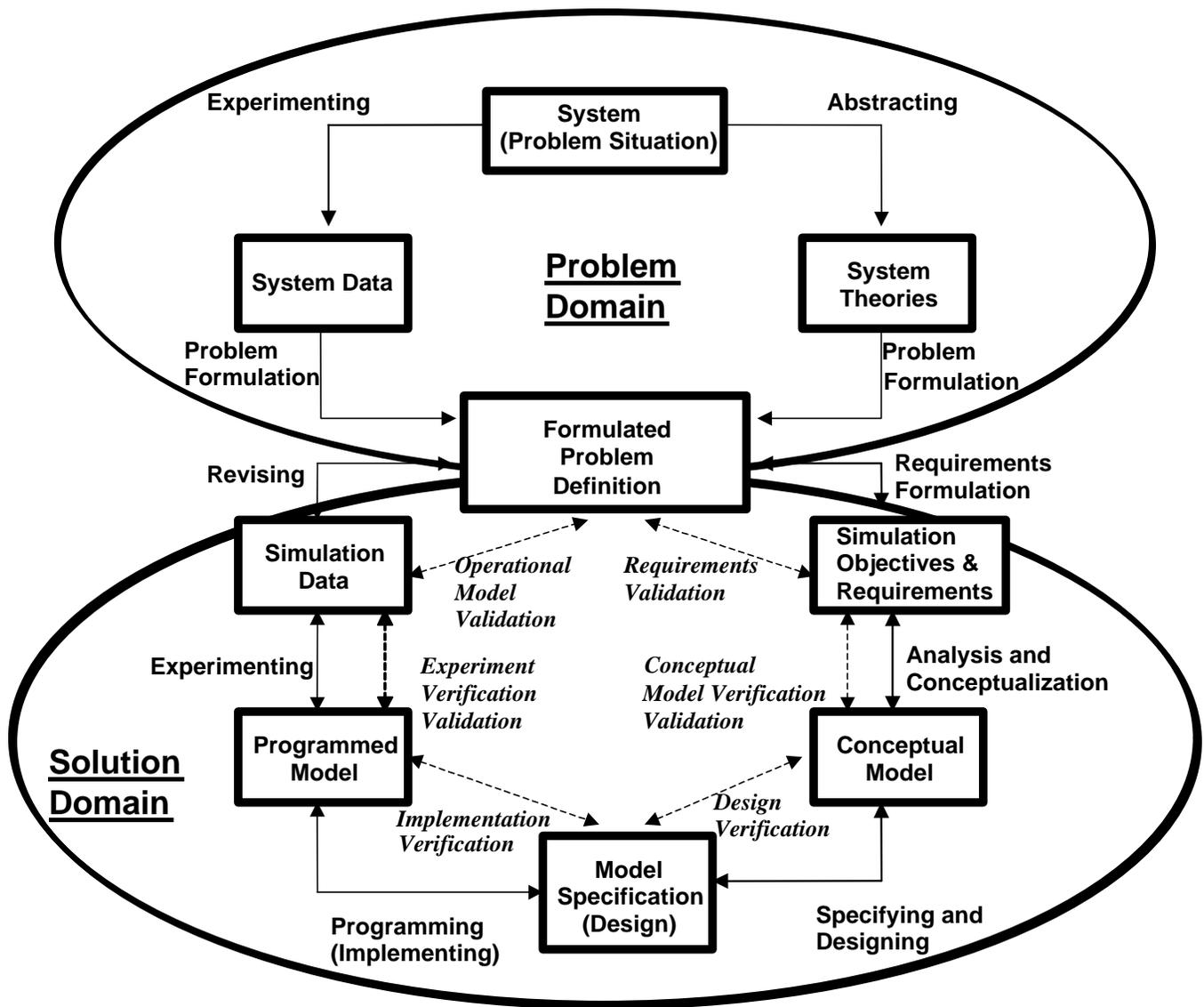


Figure 1: The Life Cycle of Simulation Study

During the evaluation and synthesis the analyst must define all externally observable entities, evaluate the content and flow of information as well as processes to establish the simulation context. The analyst creates models of the perceived simulation context to derive constraints, assumptions, and objectives that will be provided as input to the simulation concept development phase.

2.1.2 Simulation Model Conceptualization

The goal of simulation model conceptualization is to derive a simulation conceptual model. Simulation conceptual model is defined by Pace [1999] as the simulation developer's way of translating model objectives into a detailed analysis and design framework. The framework defined by [Pace 1999] characterizes the simulation concept and provides a basis from which the software, hardware, networks, and systems that will make up the simulation can be built. The simulation concept includes the representation of the mission space as well as the simulation space. All solution domain elements and specification of how they are expected to interact constitute the mission space.

The simulation space part of the simulation concept includes all additional functional and operational capabilities and elements to explain how the simulation will meet its objectives. The steps of the conceptual model development can be characterized as follows: (1) collecting authoritative information, constraints, assumptions regarding the simulation context from the simulation objectives and requirements; (2) characterizing the simulation mission space entities and processes; (3) determining and representing simulation elements constituting the simulation concept space that realizes the entities and processes of the mission space; and (4) specifying the interactions and relationships among the simulation elements.

A variety of paradigms and conceptual frameworks exist to support conceptual model development. The simulation implementation worldviews such as event scheduling, activity scanning, or process interaction as well as the domain modeling decisions are made during the model conceptualization. The accuracy and precision of the conceptual model representation depends on how the entities are abstracted. The Unified Modeling Language (UML) and Object Modeling Technique (OMT) provide a variety of concept development diagrams (i.e., use case, sequence, deployment, component, collaboration, class, state diagrams) to delineate different perspectives of a model.

2.1.3 Simulation Model Specification and Design

Model specification and design process entails the translation of the conceptual model into a detailed simulation software design model. The simulation software design can be represented by using a variety of paradigms and model representation methodologies such as structured methods, object-oriented modeling languages (i.e., UML, OMT), flowcharts, activity diagrams, condition specification [Overstreet and Nance 1985]. It is essential to choose an appropriate modeling paradigm for the application domain and the simulation conceptual model identified during the earlier phases of the life cycle. There exist additional concerns such as quality that influence the model specification and design methodology selection. Quality attributes such as

maintainability, adaptability, portability, reusability requires certain model specification methods. The need for highly critical simulations in decision making may necessitate the use of formal methods to assure that model design satisfies certain formal properties required by the simulation context. Design representation and specification formalisms that facilitate logical inference with sound proof rules may be necessary to establish the correctness of the model with respect to these formally specified properties. Hence, the application domain and the utilized paradigm for model development influence the selection of both the modeling and validation methods employed during the life cycle of a simulation study.

2.1.4 Simulation Model Implementation

Simulation model implementation is a programming or construction (i.e., model composition) process in which the model specification (design) is translated into an executable simulation. The programmed model in the life cycle denotes the executable simulation developed either with existing simulation packages (i.e., SIMSCRIPT, GPSS, MODSIM III, SIMULA, JDEVS, SIGMA) or high-level programming languages. Programming simulations with high-level programming languages requires understanding of conceptual frameworks (i.e., worldviews) of simulations. Event scheduling, activity scanning, and process interaction are the major worldviews with which model implementors need to be familiar. The choice of programming paradigm (i.e., object-oriented, procedural, object-based) has implications on the V&V process as well. The constructs of a programming language paradigm (i.e., inheritance in object-oriented development) might require proper test selection and model complexity measurement indicators to accurately assess quality of a model.

2.1.5 Experimentation and Revision

Simulation is performed to conduct experiments for the purpose of understanding the behavior of systems and evaluating their alternative operation strategies. This requires collecting the desired information from model execution and interpreting the results for decision making. Model developers need to be aware of experiment design methods to make valid inferences and collect information with minimal cost. There exist a variety of experiment design methods such as 2^k and *factorial* (full or fractional) that can be employed to predict the effect of various input variables on response variables. Detailed discussions on various simulation experiment design and analysis methods can be found in Jain [1991].

Revision or model redefinition refers to the process of modifying the model to enhance, correct, or adapt to changing contextual requirements. It is a well-known fact that almost two-thirds of model development time is devoted to maintenance evolution. In particular, the current trend towards using open standards supported by COTS in developing open distributed models increases the likelihood of constant model modernization and evolution. The usage of COTS components and the need for modernization introduce vendor driven upgrade problem. That is, as new or upgraded technologies are inserted into the model baselines, the model developers spend enormous amount of time to instill back confidence in the models connected to legacy virtual and real-time systems to assure backward compatibility.

2.2 Verification and Validation Throughout the Development Life Cycle

Verification and validation is a continuous activity throughout the development life cycle. The V&V processes are interleaved with the development processes as shown in figure 1. The V&V stages are simulation objectives and requirements validation, conceptual model V&V, design verification, implementation (programmed model) verification, experiment V&V, and operational validation. Each of these credibility assessment stages is briefly defined below. The detailed discussion on V&V technologies and methods applicable for each credibility assessment stage is given in section 2.3.

2.2.1 Simulation Objectives and Requirements Validation

Simulation requirements are derived from formulated problem definitions to characterize the simulation context and formalize the high-level operational as well as functional capabilities that are expected from the simulation. The validation of the requirements needs to assure that the authoritative information, doctrine, and strategies of the system's context are accounted for in the simulation context depicted by the simulation requirements. Specifically, system's environment and boundaries along with the characteristics of its inputs need to be taken into consideration. The influences of the inputs, the causal interdependencies, processes, and organizational characterizations need to be precisely and accurately reflected in the simulation context (i.e., requirements). Use cases, scenarios, and functional models related to the system help establish the correspondence of the devised simulation context to the formulated problem and system definition. It is also essential to check the internal consistency, completeness, coherence, and correctness of the simulation requirements. The next section overviews these evaluation criteria in detail.

2.2.2 Simulation Conceptual Model Verification and Validation

Simulation conceptual model validation is defined as determining that the theories and assumptions underlying the conceptual model are correct and the representation of the validated requirements is reasonable at the correct level of fidelity for the intended purpose of the model. In particular, the conceptual model's structure, logic, mathematical and causal relations, and the processes need to be reasonably valid and accurate representation of the real system (i.e., as defined in the simulation context). The constraints and assumptions of the system theories and context needs to be incorporated into the conceptual model. Examples of theories and assumption are stochastic processes and their constraints such as linearity, independence, stationary, and probabilistic distributions of arrivals. The conceptual models are also required to be internally complete, consistent, coherent, and correct. The simulation space indicated by the conceptual model incorporates all system entities of the simulation context in the mission space [Pace 1999]. The simulation space of the conceptual model, as defined in [Pace 1999], needs to incorporate additional essential control and operating characteristics of the simulation to satisfy the simulation requirements. Furthermore, the conceptual model should not include conflicting elements, entities, and processes. That is, entities need to be represented from a compatible perspective to facilitate consistency and interoperability with regard to potential simulation features such as attribute/component fidelity, coordinate systems, event timing. Redundant elements need to be avoided to establish a coherent concept of the simulation in which all

components have certain functions and that all model components are reachable. There exists a variety of conceptual model evaluation quality criteria such as completeness, consistency, modularity, implementation independence, maintainability, and generality [Lindland *et al.* 1994].

2.2.3 Simulation Model Design Verification

Model design verification is concerned with demonstrating the sufficient accuracy of the conceptual model transformation process that translates simulation conceptual model to a detailed design model. The way verification is perceived and deployed depends on the modeling design methodology and paradigm utilized for representation. In addition to the widely applied subjective criteria such as “level of agreement” with the “intended domain of applicability”, formal analysis, evaluation, and verification methods are necessary to measure and predict certain design quality metrics from the model. Design metrics as well as formal and semi-formal methods such as *model checking* can help model designers prove the satisfiability of certain functional as well as non-functional properties defined in the conceptual model. The notions of completeness, consistency, coherence, and correctness apply to model design verification as well.

2.2.4 Simulation Model Implementation Verification

Simulation model implementation verification is concerned with demonstrating that implementation (construction) of the design is an accurate realization of the design. That is, the entities, processes, and associated constraints and assumptions are implemented correctly. Model analysis and testing is performed to determine the degree of accuracy of the transformation from the design to the realization. Model analysis deals with the static analysis of the software or model realization to predict certain control and data-flow properties of interest. Model testing, on the other hand, is performed to demonstrate the existence of errors in the model by subjecting the model to test cases. The model is executed as a trace-driven simulation or using developed test cases to measure response variables and the involved processes. The observations are compared to the actual system data or test case oracles to determine any deviation from the expected behavior.

2.2.5 Simulation Model Experiment Design Verification and Validation

Simulation experiment design V&V entails substantiating the correctness and accuracy of the experimental designs. As mentioned above, there exist a variety of experiment design methodologies. Full and fractional factorial designs [Jain 1991], response-surface methodologies [Law and Kelton 1991], and ranking and selection [Law and Kelton 1991] are widely utilized to formulate efficient and systematic experiment designs to make valid inferences. Methods for *variance reduction* can be used to increase the statistical accuracy for a given amount of simulation. Other issues such as theoretical accuracy of random number generators, accurate implementation of random variate generators, and sufficient testing of random number generators permeate the experiment design V&V. Furthermore, it is essential to utilize appropriate statistical techniques to determine statistical significance of deviations. Consistency in replicating the identical experimental conditions in testing alternative operating strategies is essential for valid and accurate inference making from the measured results. Law and Kelton

[1991] suggest the proper handling of the initial transient start up time in simulation studies. This problem needs to be resolved and handled in simulation output collection and consideration.

2.2.6 Simulation Model Operational Validation

Operational validity deals with determining that the model's output has sufficient accuracy with respect to system data, theories, and definition for the intended purpose of the simulation study over its intended domain of applicability. Sargent [1996; 1999] defines a variety of means to determine the operational validity. Graphical comparisons of data, comparisons of means, variances, maximums, and distributions, relationships between different output variables, statistical significance and hypothesis testing, and confidence intervals are used to predict the degree of relevance or deviation of simulation data from the system data, theories, and quantitative expectations. While this perspective emphasizes statistical comparison, non-functional qualities such as causality, trends, and patterns are also essential in operational validity.

2.3 Verification and Validation Techniques, Tools, and Technologies

Figure 2 presents a taxonomy of widely known V&V techniques. The details with regard to taxonomy and techniques are discussed in Balci [1994] and Yilmaz and Balci [1997]. The techniques are classified into four main perspectives. The classification is based on the level of formality required by the techniques.

That is, the techniques lie in a spectrum of informal to formal. Informal techniques rely on subjective human decision-making using rigorous and well-defined methodologies. Static techniques are based on the model design and implementation artifacts, and they do not require execution of the model. On the other hand, dynamic analysis requires model execution and experimentation to assess credibility. Formal methods, which are at the other end of the spectrum, rely on formal process of symbol manipulation and inference according to well-defined proof rules of the utilized specification language.

2.3.1 Verification, Validation and Testing of Object-Oriented Models

Although formal verification and testing are essential elements in the success of component-based development, advances in these areas have not been widely applied or adapted for unique problems of reusable component verification. The Reusable Software Research Group at Ohio State University is one of the few research groups which has done extensive work in the field of local certification and modular reasoning about the behavior of component software. Their web site can be accessed at <http://www.cis.ohio-state.edu/rsrg>. In the next two subsections the advances in two areas related to object-oriented component-based systems verification and testing are examined: formal verification for abstract data types and object-oriented software testing. Recently, the software testing community has tackled new sources of problems introduced by the OO paradigm. [Binder, 1994; Binder, 1995; Binder 2000; Cheatham & Mellinger, 1990; Smith & Robson, 1990]. Hegazy [1989] has investigated interesting testing issues related to reusable components such as controllability, observability, and genericity. The V&V issues in object-oriented simulation has been first discussed in Yilmaz and Balci [1997].

Yilmaz [1998; 2000] extended on this work to investigate the V&V issues with regard to component-based simulations as well as their certain characteristics such as *object-flows* [Yilmaz 2001] under process interaction semantics.

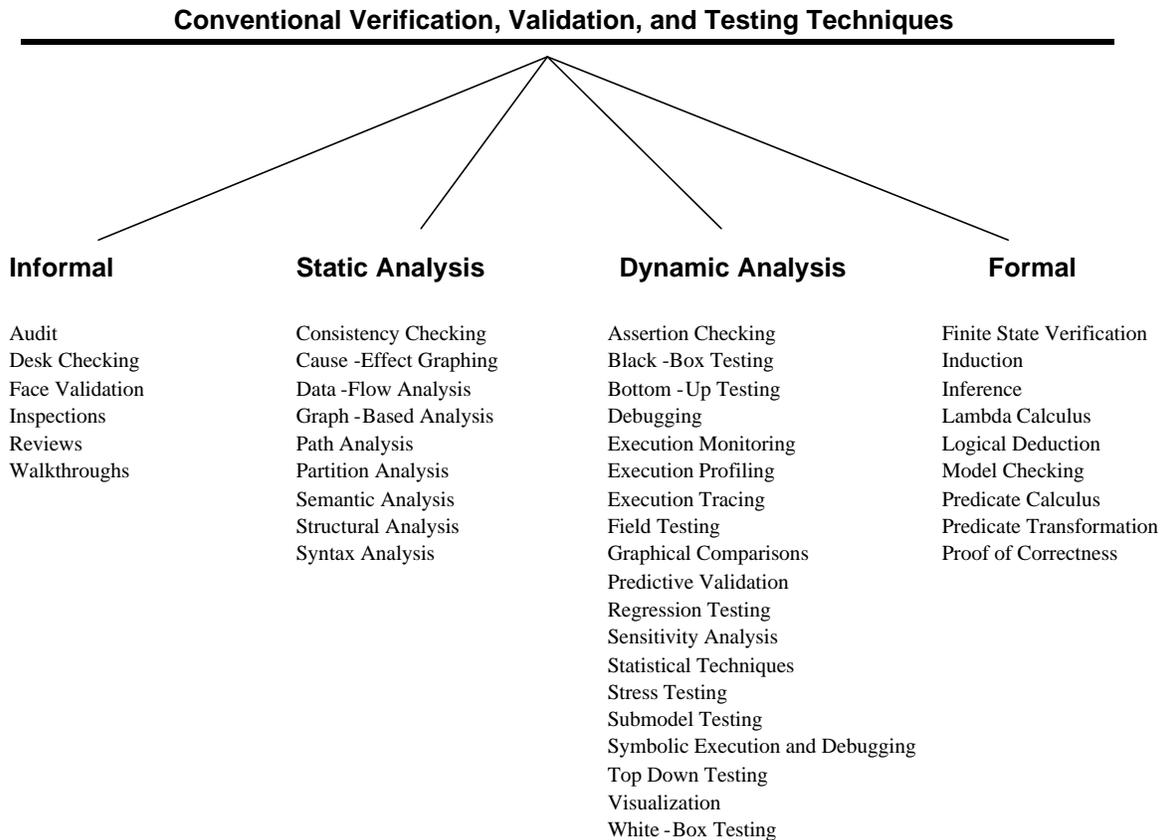


Figure 2: Taxonomy of Conventional V&V Techniques
(Modified from [Yilmaz and Balci 1997])

2.3.1.1 Formal Methods for Abstract Data Types and Conceptual Models

Since component software involves a higher risk of poor verification than does conventional software, it is essential that a reusable, concrete component is proven completely dependable and accurate with respect to its published properties. Since testing itself is not capable of proving the absence of errors, it is necessary to support it with formal verification. Furthermore, as Weide and Hollingsworth [1992] state, local certification of components is a prerequisite for tractable reasoning and verification of large software systems built using these components. Formal reasoning is based on comparing two formal objects, i.e., demonstrating the correctness of a program with respect to a specification, proving properties about a given system description.

There are two general approaches for showing the correspondence of two formal objects: theorem proving and model checking. Model checking, which works on finite transition systems, has proven to be a successful technology for verifying hardware [Clarke, Grumberg, and Long

1994; Wing and Farahani 1995]. It exploits the finiteness property of the hardware by performing an exhaustive analysis of the transition system's states. The model checking approach is also based on the assumption of finiteness of the state space of the software. Applying model checking to software may seem inappropriate at first glance due to the possibly infinite state space of software. However, based on abstractions of the state space of software, a model of the system rather than the system itself can be used, thus raising the level of abstraction at which the reasoning is performed. Therefore, while one can still fall short of exact reasoning about the original system, it is possible to perform approximate reasoning more quickly and with less effort.

The roots of formal verification based on theorem proving occur in the initial work of Floyd [1968], who first introduced the concept, Hoare [1969], and Dijkstra [1976]. Verification is based on demonstrating that given the truth of a predicate before a program node, when the node is executed, then the post-condition predicate would be true. Hoare extended Floyd's model by placing it within the formal context of predicate logic. He based this approach on a series of axioms describing the behavior of each language construct and proving, using mathematical logic, that the program has the desired pre- and post-conditions. Dijkstra developed a model similar to Hoare's axiomatic model, called *predicate transforms*, which computes the weakest preconditions and then demonstrates that they imply the given precondition of the specification. These approaches constitute the foundation of axiomatic approaches.

The use of data types, object-orientation, and modularization has led to another model called *algebraic* specifications. Specification and verification based on algebraic methods focus on the axioms defining the interactions of operators of the signature with one another. Additional works based on the concept of an Abstract Data Type can be found in Algharbi [Shaw 1981], Modula-2 [Ernst *et al.* 1982], Ada/ANNA [Luckham *et al.* 1987], and Larch/Ada [Guaspari *et al.* 1990]. Along with composition rules and a context-induction prover that can demonstrate the relationship between component specifications at different levels of abstraction, a pure algebraic approach to the specification of reusable components is shown to be useful in retrieving and identifying appropriate implementations for reuse [Cramer, Doberkat, and Goedicke 1994]. Functional correctness and verification based on operational semantics have also received attention from the formal methods community. In functional correctness a program is considered as a function from some input domain to some output domain. Assuming that it is also possible to represent the specification as a function, verification process has to show that they are equivalent functions. The operational model is based on using some more abstract interpreter and showing that the program and the abstracted program have equivalent properties. Vienna Definition Language (VDL) and denotational semantics are two examples based on the concept of operational semantics. Krone [1988] provides verification rules using the proof system of RESOLVE [Hegazy 1989].

The object-based concurrent systems community reveals the development of a trend toward providing useful perspectives for object system verification. Most of the approaches are based on some form of process calculi (i.e., CCS calculus, pi calculus). Other approaches rely on trace semantics and actor systems. The verification work involving specific OO languages such as POOL [America, 1989] and FOOPS [Goguen 1986; Goguen & Meseguer 1986] are based on Hoare-style proof techniques and algebraic techniques, respectively. Another study of object-

oriented systems verification is that of Fiadeiro and Maribaum [1994], which uses temporal logic to prove that the properties to be verified are logical consequences of the axioms of the specification.

Formal methods in component certification focus on specification and verification of components using formal languages and associated proof systems. In contrast, recent efforts have attempted to create a model-based verification process founded on selective and pragmatic application of formal methods, using simplified models that focus on error detection rather than formalized proofs [Gluch & Weinstock 1998].

By deriving test suites and new analysis methods based on partial models that capture the essence of the component under certification, model-based verification typifies a lightweight approach to the usage of formal and semi-formal methods. The models evolve during the component development, ultimately leading to the final software artifact. The method used for describing the models and for formalizing the evolutionary steps provides a basis for verifying the transformational accuracy of the overall development process.

In model-based verification, a well-founded formalism of software engineering models is chosen to capture the essential properties and behavior of the system under study. The formalism should have a well-defined language that includes mathematical and logical concepts permitting deduction and reasoning. The formal or semiformal models are then used to reason about the behavior of the component at higher levels of abstraction and to provide a basis for test suite generation.

2.3.1.2 Object-Oriented Model Implementation Testing

Many books and articles have been written on the topics of object-oriented analysis, design, and software development in general. Based on several of these, the perception about object-oriented testing can be classified as follows:

- ❑ Testing is not discussed [Martin and Odell 1992; Shlaer and Mellor 1992].
- ❑ There is no need for a new approach; testing is not affected by the object-oriented paradigm [Booch 1991].
- ❑ Object-orientation makes testing easier due to the new principles and concepts, which the paradigm introduces [Rumbaugh *et al.* 1991; Wirfs-Brock *et al.* 1990].
- ❑ Object-oriented concepts such as encapsulation, inheritance, and polymorphism make object-oriented testing different and imply new ways of testing [Berard 1992].
- ❑ Object-oriented development strategies are significantly different than traditional strategies; thus there is a need for new strategies for testing. Moreover, due to the incremental and iterative nature of object-oriented development, unit and integration strategies must be redefined and testing must be a continuous activity with particular emphasis on regression testing [Booch 1994; Jacobson *et al.* 1992].

Three different approaches to object-oriented model implementation testing are observed. These approaches are categorized by the degree and manner they incorporate other paradigms as shown

in Figure 3. A detailed discussion of these methods can be found in Yilmaz and Balci [1997] and Yilmaz [1998].

- *The conventional approach:* In this approach existing well-known techniques are used in testing without any adaptation or extension for object-oriented paradigm. In particular, these techniques are applicable at the method level testing. That is, the methods of objects are actually functions and procedures that use imperative language constructs. Thus, conventional techniques are applicable for method level testing.
- *The adaptive approach:* There is vast amount of research conducted in procedural software testing. The useful techniques borrowed from earlier research can be used in a new way by adapting or extending to include object-orientation.
- *The specific approach:* In this approach new techniques, which are based on object-oriented formalisms, are used to develop verification and validation strategies.

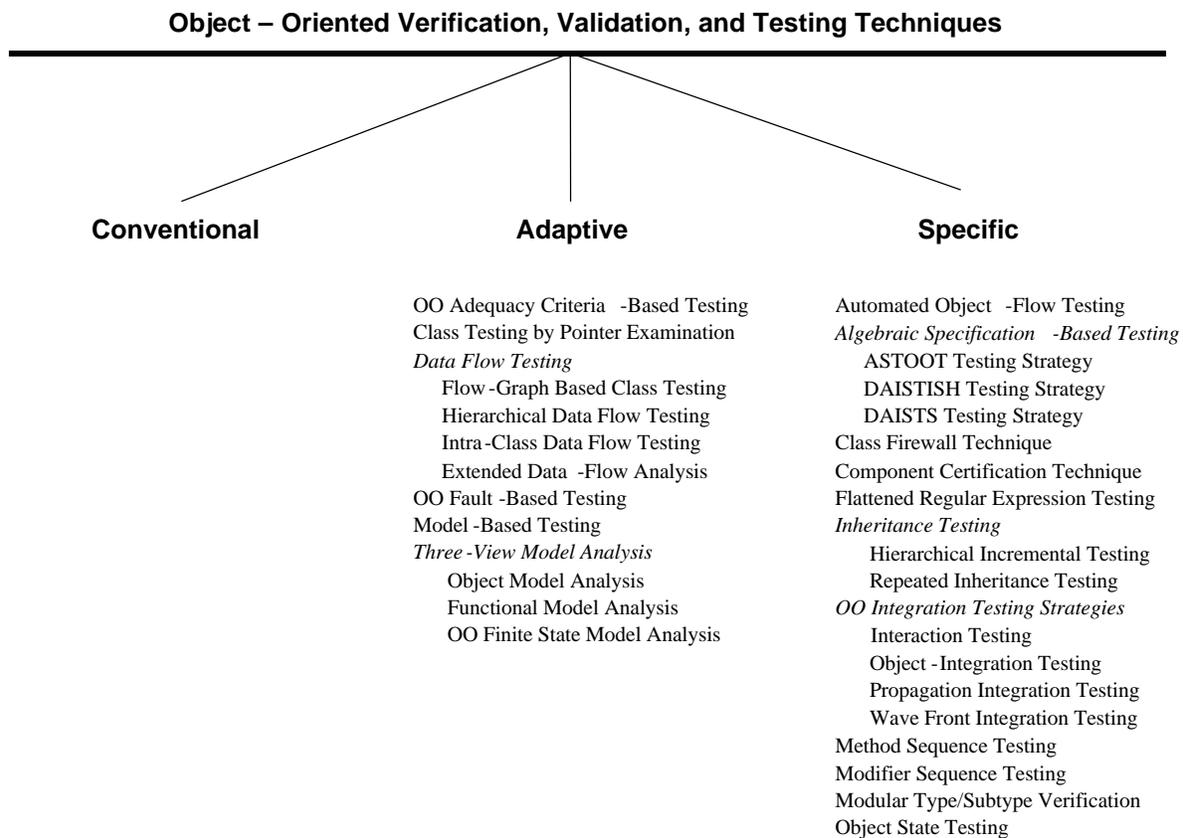


Figure 3: A Preliminary Taxonomy of Object-Oriented V&V Techniques
[Yilmaz and Balci 1997]

The differences between object-oriented and procedural software are discussed in Binder [1994]. Another of the first works that considers an object as the unit of testing and that proposes adaptations of conventional techniques is Fiedler [1989].

Perry and Kaiser [1990] discuss the theoretical view of object-oriented testing by describing the applicability of Elaine Weyuker's test adequacy criteria. Although the criteria were originally designed for procedural software, they are useful enough to determine an adequate level of testing for object-oriented software. The authors list each of the criteria and discuss the applicability to the object-oriented paradigm.

Another interesting approach to testing is the adaptation of aliasing analysis techniques to uncover errors. D'Souza and LeBlanc [1994] examine the use of pointer aliasing as a testing technique. They summarize and evaluate the existing techniques and then show that using pointer aliasing information can be as powerful in its ability to detect errors as other techniques. The basic idea involves traversing the run-time structure obtained by the test case execution, which is stored in a table of pathnames. Each entry in the table is composed of a pathname, an object identifier, and the dynamic type of the object associated with that pathname. Then the table is sorted with object identifiers. Finally, the table is scanned for duplicate object identifiers, which due to sorting appear adjacent to each other. The duplicates indicate that the corresponding pathnames alias the same object, which can be an anomaly.

One of the well known testing techniques in the context of procedural paradigm is data flow testing. Although conventional data flow testing has been successfully applied to testing of programs developed using the procedural paradigm, it is not powerful enough to be used for object-oriented paradigm [Parrish *et al.* 1993]. However, there are new adapted data flow techniques that are appropriate for the object-oriented paradigm. Parrish *et al.* [1993] argue against the use of only specification-based techniques. They propose a theory for the adaptation of conventional flow graph-based strategies to testing object-oriented programs. They develop a formal framework to model classes with flow graphs. Then analogous definitions of node coverage, branch coverage, path coverage, definition coverage, use coverage, and du-path coverage are defined on the class flow graph. Each node in the graph refers to a method of the class. The edges between nodes represent the feasible sequence of method invocations, which can be derived from the model-based specification of the class.

Hierarchical data flow testing technique, which is an extension of the existing data flow techniques, is proposed by Subramanian, Tsai, and Kirani [1994]. The authors decompose an object-oriented program into three different hierarchies: classes, objects, and attributes. They describe class flows, object flows, attribute flows and then define the interrelationships among them. Definitions such as "defined," "referenced," and "killed," which are used in traditional data flow testing, are extended for class, object and attribute flows. The analysis is based on an algorithm with two phases. In the first phase class flow, object flow, and attribute flow are performed; in the second, any ambiguities raised during the class and object flow stages are resolved. As do the authors of previous approaches, Subramanian, et al., show the usefulness of the technique for method sequence anomalies. One of the most important contributions of this approach is its shifting of focus from variables to objects and classes; thus anomalies at this level of abstraction can also be revealed.

In Chen and Low [1995] an extension of traditional data flow analysis is proposed. The technique promoted by the authors uses the memory location of variables and class members

rather than the improper sequence of actions on a variable. This procedure allows for the detection of anomalies based on pointer variables and class operators.

Another technique for adapting data flow testing to object-oriented paradigm is discussed in Harrold and Rothermel [1994]. They argue that most of the existing approaches are based on specification-based testing techniques and that there is a necessity for augmenting these approaches with code-based or white box techniques. Their technique defines three levels involved in performing data flow class testing: (1) *Intra-method testing*, which performs data flow testing for individual modules; (2) *Inter-method testing*, which tests the interaction of methods inside a class; and (3) *Intra-class testing*, which tests the sequence of calls that can be initiated by the user of the class..

Most of the approaches suggested for object-oriented testing do not mention the domain structure and rules for which the model is developed. However, the structure of the domain and the knowledge structure represented by the relationships among objects also require verification. For instance, Addy [1997] discusses the importance of verification and validation in the domain engineering phase of reuse-based software engineering. Kandelin and O'Leary [1995] extend their previous research on rule-based system verification to develop verification approaches for object-oriented AI models. Their approach uses meta-knowledge from the domain to examine the completeness, consistency, correctness, and redundancy of the object model.

By identifying faults specific to the object-oriented paradigm, one can develop testing strategies either by adapting conventional techniques or by developing pure object-oriented approaches. Hayes [1994] concentrates on the necessity of developing fault taxonomy to guide testing. She examines several fault types and associated verification techniques that are mentioned in Firesmith [1993] and Purchase and Winder [1991]. Hayes argues that in order to adapt conventional techniques it is necessary to understand the specific components of an object-oriented system.

Hayes [1994] identifies five significant components: (1) objects, (2) associated messages and methods, (3) class hierarchies (inheritance, aggregation, association relationships), (4) external interfaces, which are connections to databases, users, communication channels etc., and (5) tools and utilities. Tools and utilities are regarded as highly usable components. The author argues that these components require a certification procedure, since they will be reused in a variety of systems without customization. The methodology developed by Wohlin and Runeson [1994] provides a useful technique, based on usage modeling, for certifying components of a model.

Firesmith [1993] analyzes the problem of testing and recommends a hierarchical testing strategy based on the existing techniques. He identifies both the differences between the object-oriented and procedural paradigm and their implications for testing. Based on the levels of testing (i.e., object, class, subassembly, message, exception, scenario), taxonomy of errors and their priority for testing are presented. Then he suggests a verification technique for each error type listed in the taxonomy. The techniques proposed are conventional black-box testing, white-box testing and inspections to detect these specific errors. However, Firesmith does not explain how to apply the techniques to uncover errors.

There are other approaches that focus on test case generation and testing using different views of the system: object model, dynamic model, and functional model. Usually these techniques are influenced by the Object Modeling Technique (OMT) discussed in [Rumbaugh *et al.* 1991] for the design and modeling of object-oriented systems. Thus the test cases are prepared based on three different models: (1) Object Model, (2) Functional Model, and (3) Dynamic Model.

Poston [1994] discusses a project in which automated testing tools are integrated with the OMT life cycle. The object model identifies the classes, attributes and their relationships (i.e., inheritance, association, aggregation). Although the object model incorporates name, type, and initial values of attributes, it does not include domain descriptions. The testers add textual annotations to the object model to make it test-ready. The dynamic model introduces graphical notations for the specification of events, states, transitions, and hierarchies of states. However, creating a test-ready dynamic model is hard and time consuming. After creating test-ready models, a test case generator must be used to develop cases based on such design techniques such as boundary value analysis, equivalence class testing, cause-effect graphing, and event-based testing.

Kung, Hsia, Toyoshima, Chen, Kim, and Song [1995] developed a methodology that uses a test model composed of three views of the system. Those views are represented by three types of diagrams: 1) the object relation diagram (ORD), 2) the block branch diagram (BBD), and 3) the object state diagram (OSD). Those diagrams are extracted from the source code using a reverse engineering technique [Kung, Gao, Hsia, Toyoshima, and Chen, 1993]. The ORD diagram represents aggregation, association, and inheritance relationships. The BBD deals with those aspects of the system concerned with the control structure, transformation of values, and functional dependencies. The OSD considers the dynamic model of the component [Kung, Suchak, Gao, Hsia, Toyoshima, and Chen, 1994]. The usefulness of the test model is summarized in Kung, *et al.* [1995] as follows:

- ❑ The tester and maintainer can understand the structure and relationships among the components of an object-oriented program.
- ❑ The model allows the tester and maintainer to find strategies based on the test model to reduce testing effort.
- ❑ The model facilitates the definition and analysis of test criteria..

Another approach advocated in Software Architects' Synthesis model (SASY) by McGregor and Korson [1994] is part of a comprehensive testing process utilizing the iterative OO development approach. The authors represent SASY as a tool for structuring the development process and for allowing the integration of the testing process during each iteration stage. Three main categories of testing are identified: testing the development process, testing analysis and design models, and testing implementation. McGregor and Korson focus on and suggest strategies for the last two categories, namely, testing the implementation and models developed during the analysis and design phases. The system model testing process is based on the completeness, consistency, and correctness criteria. Use-cases are identified to develop interaction models, which are checked for errors, ambiguities and omissions. Class-Responsibility-Collaboration (CRC) cards are prepared for modeling individual classes during the specification phase. Then several checks are

performed on these CRC cards to demonstrate their correctness and consistency with respect to the object model.

The testing proceeds by using the models of OMT; that is, object model, dynamic model, and functional model. During this phase not only the accuracy of the transformation from the analysis model to design model must be verified but also the consistency of these different views must be considered. For the system model testing process the authors propose guidelines for testing the three different views of the system: object model, functional model, and dynamic model.

The testing process usually involves the generation of cases which have input and expected output pairs, such that the program processes the input and then compares the actual output to the expected output (i.e., oracle) in order to determine whether the test case is successful or not. Object-orientation represents a significant departure from conventional process-oriented models in which there is a clear division between process and data. In the process-oriented model the data is static, and complex processes reflected by large procedures represent the domain-dependent semantics. By contrast, object-oriented modeling encapsulates both the data and its domain-dependent semantics. Smith and Robson [1990] also argue this modeling difference in order to demonstrate that the testing process defined above is not suitable for object-oriented paradigm.

Algebraic-Specification-Based Testing approaches use algebraic specifications to object-oriented testing. One of the earliest systems to address the question of testing data abstractions is DAISTS (Data Abstraction Implementation Specification and Test System) [Gannon *et al.* 1981]. In DAISTS the axioms of an algebraic specification are used as an oracle for the testing implementations of the Abstract Data Type (ADT). A test case consists of a set of arguments for the left-hand side of the axiom. The test case is given as input to the left-hand side and right-hand side of an axiom and then the results are checked by a user-supplied equality (EQN) function.

Doong and Frankl [1994] describe an approach called A Set of Tools for Object-Oriented Testing (ASTOOT) that is based on the algebraic specification of ADTs. ASTOOT consists of three components: driver generator, compiler, and simplifier. Under this system, the user utilizes an interactive test generation tool (based on the compiler and simplifier) to create a sequence of operations from algebraic specifications. The simplifier then uses rewrite rules provided by the compiler to produce two equivalent sequences of operations that form a test case. Then the driver generator executes the test cases.

A more recent approach proposed by Hughes and Stotts [1996], called DAISTISH, involves a perl script which uses the formal specification of an ADT; it combines this specification with the actual code and then creates a test driver. DAISTISH is similar to DAISTS; however, DAISTISH is more effective in creating test drivers for languages that use side effects to implement ADT. This approach is also similar to ASTOOT; however, the authors concentrate on the semantic-specific issues of correctly duplicating the objects, rather than on automating the test case selection problem. The differences between these tools are summarized in Hughes and Stotts [1996].

Also worth mentioning are the retesting and class firewall concepts developed by Kung *et al.* [1995]. These programs undergo continual changes. By the use of functional decomposition principles, the procedural paradigm provides a well-defined tractable structure. Hence the effects of a change can be traced easily. However, in the case of object-oriented software, the effects must be propagated to all the objects with relationships to the object under change. The Object Relation Diagram (ORD) and Block Branch Diagram (BBD) are used for identifying the regions of the code in need of retesting. The details of constructing the class firewall, its use in retesting, and an algorithm for creating an optimal test order for the firewall are described in Kung, Gao, Hsia, Lin, and Toyoshima [1995].

Reuse is one of the key issues in object-oriented development. Classes, components, and highly reusable model libraries can be reused for different problem domains without wasting the effort for redeveloping the same functionality. However, since these components will be reused in a variety of systems without customization, they require a certification procedure. The methodology developed by Wohlin and Runeson [1994] provides a useful technique, based on usage modeling, for certifying components of a system. The objective of their method is to obtain reliability measure for the components. During development for reuse, they suggest that components can be certified with usage testing by developing usage profiles and then applying a certification model that provides the level of confidence in them. They generate test cases by using a hierarchical usage model by a state hierarchy model which uses Markov chains to model the state transitions. They attach probabilities to usage and behavior transitions and then use these measures to develop test sequences. The test sequences facilitate the collection of failure data and provide a basis for the certification measure.

Harrold, McGregor, and Fitzpatrick [1992] discuss an incremental approach for the testing of class hierarchies. They suggest an order for testing the inheritance hierarchy that starts from the top. That is, testing starts with the base classes and continues down the hierarchy. The testing process progresses by testing features in isolation followed by testing the interactions among features. The authors' algorithm uses test histories from the parent classes, which are incrementally updated with information about the derived class's differences from the parent class. Each new test history allows the tester to differentiate the reusable test cases from the parent class, along with any attributes for which new test cases must be generated. The authors' approach supports three levels of attribute visibility: hidden, accessible to derived classes, and accessible to all classes. They model the inheritance concept in terms of a class and a modifier, resulting in a derived class.

Chung and Lee [1992] offer another study related to inheritance. The authors show that when repeated inheritance occurs errors may arise as a result of name-confliction. The authors build a directed graph of the inheritance structure where nodes denote the classes and edges denote the inheritance. The root classes are traversed using breadth-first traversal, and then parent classes are added to the ancestor sets of the child classes. The ancestor sets are examined to find the repeated inheritance structures, which are then classified hierarchically based on their Euler region numbers (r). Then these hierarchies are used in testing as follows: URI(1): every class in a repeated inheritance must be tested at least once; URI(2): every repeated inheritance with $r = 2$ must be exercised; and URI(3): every closed region with $r = 3$ must be exercised. The authors

propose that after such hierarchies are discovered, McCabe's cyclomatic testing strategy should then be applied to them.

The complexity of the object-oriented paradigm arises from the dynamic and diverse patterns of interactions and collaborations among the objects. Testing individual methods and classes is not enough; integration of class methods (i.e., intra class integration testing) and of methods of different classes via association relationships (i.e., inter-class integration testing) is also necessary. Unlike that of procedural and functional models, object-oriented models do not possess an easily tractable system decomposition structure. Thus, top-down and bottom-up integration strategies are not very useful. Inheritance and aggregation relationships and dynamic binding thus pose challenging integration-level problems for the object-oriented paradigm. Jorgensen and Erickson [1994] discuss this problem of structure and behavior in the case of object-oriented software. They argue that the event-driven nature of object-oriented software forces a declarative spirit on testing. The two constructs proposed for the integration level of testing are as follows:

- ❑ **Method/Message Path (MM-Path)**: a sequence of member function invocations which starts with a method and ends when a method which does not issue any messages is reached.
- ❑ **Atomic System Function (ASF)**: an input event followed by a MM-Path, which ends with an output port event.

These constructs provide a framework for integration testing. However, as the systems become larger and more complex it becomes necessary to formally specify and capture the object interactions through MM-Paths.

McGregor and Korson [1994] also provide an integration approach in their testing process model. They identify two levels of interactions: those among the methods of a class and those among the methods of different classes due to associational relationships. They suggest investigating a pair of methods that directly or indirectly affect each other to produce incorrect results. The strategy proceeds by constructing test cases that identify attributes modified or accessed by two or more methods. Typically, each pair of such methods should be executed. However, in the cases of polymorphism and dynamic binding, the definite inter-class relations cannot be identified. The authors discuss applying orthogonal array-based techniques to utilize possible interactions and create corresponding test cases. However, as the order of interactions increases, the management of the technique becomes difficult.

Graham, Drakeford, and Turner [1993] suggest using black-box and white-box testing for individual methods. They recommend an incremental strategy similar to the approach discussed in Harrold, *et al.* [1992] for reducing the number of test cases that need to be re-run during the testing of inheritance structure. For object integration testing, an object instantiation tree that shows which objects can be instantiated from which others allows the tester to provide a top-down testing order. This strategy creates a plan for the integration of program units.

One of the problems of object-oriented testing is the state dependent behavior of objects [Binder 1995]. A state-based approach is given in McGregor and Dyer [1993]. The test strategy relies on the development of a state model of a class from OOA/D specification or by examination of the

implementation of the class under test. Each class keeps an invariant and each test case checks whether the current state complies with the invariant. They argue that due to polymorphism and dynamic binding, possible number of client objects and polymorphic servers can be large and they propose the use of orthogonal arrays to facilitate selection. Turner and Robson [1993] propose a new technique, called state-based testing, for the validation of object-oriented systems. They mention that due to the interactions between the features and the state of the object, new techniques are necessary. Their approach defines the state of an object as the combined values from all of its data members. Once the states are identified, the methods and the input and output states are examined. That is, the input states for which the method is applicable and the output states which result by the application of the method are identified. Then test cases are generated to create the initial condition (starting state for a method) along with the code which validates the starting state. After this, the method is applied and the resulting state is compared to the expected final state.

3. Verification and Validation Issues and Myths

The application of V&V throughout the life cycle of a simulation study is widely accepted in the simulation modeling community. However, most simulation practitioners see the life cycle as a sequential process, during which their favorite V&V tools and methodologies can be directly applied. On the contrary, different development methodologies and paradigms need to be used for different simulation development paradigms. Hence, proper selection of V&V methods is essential to accurately measure the validity and quality of model artifacts developed under that specific paradigm. For instance, model testing methods used for structured model design and implementation may not properly apply to object-oriented or component-based development artifacts. A different paradigm not only utilizes different artifacts organized in a unique structure, but also builds on slightly different life cycles and development processes.

3.1 Verification and Validation Issues with Modern Development and Implementation Paradigms

The generic simulation development life cycle presented in section 2 includes the most common phases of a process. There exist, however, methods such as agile processes that require considerable change in the organizational deployment of these phases.

3.1.1 Traditional Development Models

Schulmeyer and MacKenzie [2000] and Rakitin [2001] review and evaluate several software development life cycle (SDLC) models from the point of view of verification and validation processes.

The classic *waterfall* model is based on a well-defined sequence of phases involving requirements analysis and specification, design, implementation, and test and integration. The *modified waterfall* assumes the same steps, but allows for iteration in any particular phase back to a prior phase. Both of these models assume that a usable software product will not exist until the end of the development process. The *build* (or “incremental build”) model allows for the production of intermediate software products that incorporate increasing functionality. Once

verified, these “builds” can serve as platforms for the integration and testing of new functionality, and they can often provide users a partially functioning system that fulfills at least some of their requirements. The *waterfall* and *build* models both assume that it is possible to capture and specify system requirements when a development project begins. The *spiral* model [Boehm 1988], however, recognizes that with complex systems it is often difficult to fully specify requirements in early stages of the SDLC. This model allows for a continuous process of refining requirements, validating them through the use of prototypes, assessing risk, and refining plans - all leading to ultimate product development.

The *waterfall*, *build*, and *spiral* models usually result in the generation of artifacts, such as requirements specifications, design documents, test plans, and test procedures that are available at predefined milestones in the development process. Traditional V&V methods, including requirements tracing, specification reviews, design reviews, code inspection, and test monitoring, depend in part on the examination and evaluation of these artifacts [Lewis 1992].

The international standard *ISO 12207* is somewhat unique in that it defines overall activities for primary, supporting, and organizational life cycle processes (including V&V) without dictating the use of a specific SDLC model. Thus there could be variability in the nature of the products available for V&V. *IEEE/EIA 12207.1* is more specific in that the standard provides guidance for a series of “information items” that should be produced as a result of various processes, activities, and tasks in the life cycle [IEEE/EIA 12207.1-1997]. Many of these items would support V&V.

Additional development paradigms are discussed in Rakitin [2001]. *Model-based development* techniques are particularly useful in designing and implementing client/server and Graphical User interface based systems. (“Model” in this context refers to the use of a business process model, or application domain model, in analyzing requirements and developing the system design). *Fourth generation* life cycle models are hybrids that assume that large development efforts can use different techniques, depending on which is most appropriate, for different parts of the system. There is no guarantee that *model-based development* or *fourth generation* life cycle models will result in the full set of artifacts that supports traditional V&V.

Table 2 shows several characteristics of various life cycle models. This simplified comparison is based on factors that might present challenges for V&V. For example, in models that allow for incremental requirements definition or iteration, verification is complicated by the need to maintain a system for continually assessing requirements completeness and the impact of changes. Similarly, final validation from a user’s point of view is more complex in situations where intermediate products can be deployed in an operational environment.

Table 3 lists opportunities and risks - the strengths and weaknesses - of different life cycle models under a variety of programmatic conditions. Depending on the nature and level of uncertainties that are involved, some models may be better suited to a particular project than others. In this context, V&V must also be flexible.

Table 2. Life Cycle Model Comparisons

Model	All Rqmts Defined First?	Multiple Builds?	Use Interim Products?	Remarks
Waterfall	Yes	No	No	Recursion / Iteration, As Needed
Modified Waterfall	Yes	No	No	
Build	No	Yes	Yes	Build N = Build (N-1) + More Capabilities. Recursion / Iteration, As Needed
ISO 12207	No	No	No	Recursion / Iteration, As Needed
Spiral	No	Yes	Yes	Build N = Build (N-1) + More Capabilities. Recursion / Iteration, As Needed
Model-Based	No	No	Yes	Recursion / Iteration, As Needed
Fourth Generation	No	Yes	Yes	Build N = Build (N-1) + More Capabilities. Recursion / Iteration, As Needed

[Schulmeyer and MacKenzie 2000, p. 39]

Table 3. Life Cycle Model Opportunities and Risks

Factors	Opportunity	Risk
1. Requirements not well defined.	3,5	1,2,4,6,7
2. System too large to do once.	3,5,7	1,2,6
3. Full capability needed at once.	1,2,6,7	3,5
4. Partial capability needed early.	3,5	1,2,6,7
5. Phase out of old system to be gradual.	3,5	1,2
6. Rapid changes in requirements anticipated.		ALL
7. Rapid changes in technology anticipated.	6,7	1,2,3,4,5
8. Long run staff / funds commitment doubtful.	3,5,7	1,2

Legend - Models:

1-Waterfall; 2-Modified Waterfall; 3-Build; 4-ISO 12207; 5-Spiral; 6-Model-Based; 7-Fourth Generation

[Schulmeyer and MacKenzie 2000, p. 39]

3.1.2 New Development Models

Newer SDLC models include *rapid application development (RAD)*, *operational prototyping*, and *agile methods*. *RAD* is most effective in situations where requirements are constrained and well defined and where the project can be completed in a few months [Pressman 2001]. *Operational prototyping* utilizes rapid prototyping techniques to evolve both the product specification and design in parallel [DeGrace and Stahl 1991]; the difference here is that the prototype is intended to evolve into a finished operational product, whereas most prototypes are meant to be “throwaways.” Therefore engineering discipline must be maintained and operational standards must be observed.

The *agile methods* model, as exemplified by *Extreme Programming (XP)* [Beck 2000], extends on these evolutionary development processes. *XP* is characterized as being most effective on small to medium sized projects where requirements are vague or ambiguous. Development staff

is often divided into teams of two programmers each, and code is rapidly written and frequently compiled, executed, and integrated as part of a continuous build and test process. The objective is to “put a simple system into production quickly [and] release new versions on a very short (two-week) cycle” [Paulk 2001, p. 20].

Recent discussion about the theory and practice of XP, and *agile methods* in general, revolves around several perceived limitations. First, the issue of scalability has many arguing that these methods cannot be effectively applied to projects requiring large development teams. (Although Aoyama [1998] describes a case where a well disciplined and well managed Agile Software Process was successfully used on a project that over time involved approximately 500 developers). Second, there are some concerns about the amount of freedom XP gives developers to change architectural design during the course of the project [Paulk 2001]. And third, in the case of mission critical applications there is a question as to whether XP-like methods encourage the use of the managerial, planning, and systems engineering processes needed to produce high reliability software. Boehm [2002] summarizes many of these arguments and proposes that organizations should be flexible and familiar with both agile and more traditional SDLC models and be prepared using risk assessment to adopt the suitable approach for a given situation.

New models present challenges to traditional V&V. A deliberate attempt to reduce the amount of documentation to a bare minimum will likely increase programmer productivity [Maurer and Martel 2002], but it will also reduce the number and quality of documents that support typical V&V tasks. Furthermore, these models are schedule driven and require a tight integration of design, code, and system test activities - often conducted by the same individual. Thus, in an environment where independent verification (as discussed in section 3.2) might be most beneficial for product quality, it becomes more difficult to perform because of schedule constraints and reduced access to the testing process.

3.2 The Role and Rationale of Independent Verification and Validation

From an organizational perspective, Arthur and Nance [2000] maintain that independent V&V is necessary to establish *technical*, *managerial*, and *financial* independence. *Technical independence* requires the credibility assessment and risk mitigation efforts to utilize personnel who are not involved in the development of the system. It helps detect errors overlooked by those too close to the solution. *Managerial independence* requires that the responsibility for the V&V and credibility assessment effort be handled in an organization separate from the development organization. Hence, the credibility assessment results, anomalies, and findings can be submitted without any limitations and undesirable pressures from the development organization. *Financial independence* requires that control of credibility assessment budget be handed in an organization independent of the development organization. This independence prevents situations where V&V effort can not complete its analysis within the release time because funds have been diverted or adverse financial pressures or influences have been exerted. The benefits of independent V&V are direct consequences of maintaining technical, managerial, and financial independence [Arthur and Nance 2000; Lewis 1992]. Major benefits can be characterized as follows:

- **Benefit 1:** Independent V&V promotes objectivity through unbiased technical viewpoint.

- ❑ **Benefit 2:** Independent risk management promotes earlier detection of model and software errors during the conceptualization and design phases. As a result of earlier error detection effort and cost reduction can be achieved in removing those errors.
- ❑ **Benefit 3:** Enhanced operational correctness and statistically significant reduced variability in the development process.
- ❑ **Benefit 4:** Independent risk management team help get things right in the first place by designing testability and reliability through its assurance role. It also makes sure that the product is independently evaluated through its assessment role. Both of these roles are played throughout the development life-cycle.
- ❑ **Benefit 5:** Developers focus on getting software to work. They may lack the knowledge and training necessary to assess and improve the quality, safety, and reliability of systems. Independent risk management closes the gap of risk management knowledge.

3.3 Myths about Simulation Model Verification and Validation

Applications of model validation methods are considered difficult, expensive, and time consuming. Based on our experience and case studies in real-world examples, we challenge existing common myths about simulation model validation.

Myth 1: Model V&V can guarantee correctness, absolute validity, and credibility: The fact is that complete model testing is not possible due to large state space of involved model designs and software implementations. Model testing is known to indicate the presence of errors as opposed to their absence. Formal methods can be used during the early phases of the lifecycle to prove certain properties of models. However, complete functional formal verification of model implementation is not possible. This is partly due to the model development technologies and practices that make modular sound reasoning and inference about implementations impossible. Furthermore, functional correctness is a necessary but insufficient condition for validity and credibility. The incorrect formulation and definition of problems can also lead to acceptance of invalid models that are exhaustively tested. This is similar to the situation that leads to the necessity of validating the knowledge base in Knowledge Based Systems [Schulmeyer and MacKenzie 2000, pp. 262-285]. Inappropriate usage of V&V methods along with the lack of techniques for specific paradigms aggravates the problem. In addition, model validation has its roots in theory confirmation. The philosophy of simulation validation is, in general, in conformance with the pragmatist philosophy of science and builds on systems thinking perspective. In this view, V&V is a confidence building activity and involves subjective criteria regarding the usefulness of a model for its intended purpose over its intended domain of applicability. Since, the simulation model is built with respect to the study objectives for the identified domain, its credibility needs to be judged against the conditions for which it is tested.

Myth 2: Model V&V deals only with substantiating the accuracy of simulation models: The fact is that the credibility and validity of today's complex simulation systems require certain qualities to exist along with accuracy. The acceptability and usability of simulations are effected by quality characteristics such as maintainability, conformity, performance, and portability. In addition, the controllability and observability of models increase their testability. A testable model is likely to reveal hidden errors; hence, the arguments on its validity is more convincing.

Therefore, model V&V deals with quality as well as accuracy of models, since they are both necessary for building confidence in the model.

Myth 3: Models can be considered valid and credible as long their behavior is an objective reflection of the system entity: The fact is that a model needs to be considered as a representation of the real system that embodies a theory about how the system works from the perspective of the purpose of the study. Hence, substantiating the operational validity based on model's black-box behavior under certain experimental conditions is necessary but insufficient for its acceptability. That is, achieved accuracy in emulating the input/output behavior may be due to incorrect reasons embodied in the causal relational structure among the simulation elements and processes. Therefore, establishing the fact that the model behavior is achieved the way it is intended is also necessary for many simulation application domains (i.e., system dynamics).

Myth 4: Model V&V requires highly trained statisticians and mathematicians: The fact is that even though there exist V&V methodologies that incorporate techniques such as formal verification, model checking, and statistical inference that require mathematical logic and statistics knowledge, V&V is extremely procedure driven and methodological activity. The necessary statistical and mathematical knowledge is not often used in the derivation of mathematical properties of models, but rather application of routine procedures on the collected data. Furthermore, for advanced formal techniques, there exist tool support to guide and provide heuristics to complete even the formal and advanced mathematical validation activities. Besides, CASE and testing tools almost always support software testing and analysis phases. The widely used techniques such as reviews, walkthroughs, and inspections are informal methods that require critical thinking and analysis capabilities as opposed to using statistical inference or mathematical logic.

Myth 5: Model V&V always increases the cost of the development: The fact is that with continuous application of V&V (especially IV&V) throughout the development life cycle, the costs of a model development effort containing no IV&V component is comparable (most of the time higher) to the same development effort with an IV&V component [Arthur and Nance 2000]. As reported by Lewis [1992], the cost of IV&V is offset by the reduction in reported problems (i.e., poor requirements, lack of plans, failure to detect earlier at the requirements and specification development phases).

Myth 6: Model V&V always delays the development cost: The fact is that the strongest indicators of project failures and schedule slips are poor requirements, failure to communicate with the customer, and poor allocation of resources [Arthur and Nance 2000], which are mitigated by IV&V activities Arthur and Nance argue, with the support of a reasonably recent Standish survey [Standish 1995], that 11% of surveyed projects exceeded their time estimates by 200%, 46% of projects exceeded by 100%, and 66% exceeded by 50%

Myth 7: A simulation model is built and validated with respect to certain study objectives, therefore, it can not credibly be reused in a new context: The fact is that complete matching of model objectives and model implementation interfaces is not necessary for the reuse of models in a new context. Technically, models can be syntactically adapted or wrapped to behave

in conformance with its new context. From the objectives point of view and purpose of the study, as long as the model is *practically* useful and relatively *meaningful* when adapted for conformity to the new context, a trade-off can be made between a reuse and development from scratch decision. After all, since simulation model validity means “adequacy with respect to a purpose”, validation needs to have qualitative and subjective evaluation of components. Note, however, research for guidelines and methods are necessary to measure the degree of relevance of a model to new context’s requirements and objectives. It is widely accepted that, driven by fiscal constraints, increasing pressure exists to explore cost effective methods for simulation model development activities. The shorter timescales required as well as the need to rapidly incorporate simulation elements demanded by model-integrated applications bring new challenges. In particular, cost reduction in model development using parameterized designs with compositional modeling is one of the core problems that needs to be addressed.

Myth 8: Independent model V&V (IV&V) is not necessary since compliance to process models such as CMM guarantees high quality and credible models: The fact is that compliance with process models does not necessarily result in high-quality products. The process models standardize and streamline the development effort and enable repeatable and measurable product development. Throughout 1990s, process models and improvement strategies have been advocated as a silver bullet to help achieve high quality software intensive systems and simulations. The emphasis on conformance to process standards without considering specification conformance is likely to result in more failures. Besides, the customers are not paying for the process, but rather for the final product. Having a well-defined process in place is definitely a good idea, and it is also essential for managing large complex software intensive systems. However, unless product quality is sought throughout the process, process quality itself can not deliver the high quality and credible simulations.

4. Emerging Prospects for Model V&V Research

A close look at the model validation history indicates that quality and validity of models have been interpreted in different ways based on the mission context and the types of problems being solved. Evolution of modeling paradigms, methodologies, tools, and technologies had considerable impact on the way V&V is perceived. These advances are partly influenced by the changing nature of the problems attacked as well as the emergent paradigms (i.e., object-oriented, agent-based) used by the modeling community. In this section we discuss the emerging prospects for future V&V research based on the prominent new paradigms and methodologies.

4.1 Formal Methods for Model V&V

Formal methods, as discussed in prior sections, are mathematically based techniques that provide frameworks within which model developers can specify, reason, and verify about designs and simulations in a systematic manner. Formal methods in software are, to a certain extent, applicable to simulation software as well as model designs. Such methods can be used to reveal ambiguities, incompleteness, inconsistencies in system theories and models that simulate them. Furthermore, formal methods with sound inference and reasoning mechanisms help model developers prove certain properties about their models. Since formal methods enable systematic demonstration of the correspondence of the model with respect to its specification, the

conceptual model validation process can benefit from such sound inference and reasoning methods. That is, the deductive apparatus of formal methods can help model developers to establish the correspondence of their conceptual models to system abstractions, theories, and properties depicted in a formal specification language. Developing specific properties of interest for certain application domains and introducing approaches that satisfy model conceptualizations such as event graphs and DEVS formalisms are promising areas of research that will introduce formal methods to early phases of the simulation lifecycle.

4.2 Model V&V with Reuse: Connections with Interoperability

Simulation development with reuse can not only significantly reduce effort and cost but also improve the quality of simulations. Reusing existing simulation components requires reasoning about their fitness and suitability in the new context. Syntactic interface compatibility is a necessary but insufficient condition for reuse and composable simulations. Semantic compatibility of simulation components is essential to alleviate differences in fidelity levels, representation granularity, time management, and data compatibility. Establishing semantic compatibility among diverse components under a new federation (i.e., High Level Architecture) requires meaningful and timely exchange of information. Furthermore, original objectives of a reused simulation component may not be in conformance with the needs and goals of a new simulation. Verifying and testing components to assess their fitness into a new formally specified context requires methods that detect mismatches among specification. Since complete matching is not a practical solution, fuzzy and approximate matching with formal decision methods are more likely to be useful. Building models for reuse introduces new issues such as general applicability and conformance to published specifications. That is, each model needs to publish its properties as well as its functional and non-functional behavior. The verification of a model component has to demonstrate that the component is correctly developed with respect to the published specification such that as long as the context uses the component in conformance with its specification, the reused component is guaranteed to deliver published behavior. Yilmaz and Edwards [2002] illustrate this viewpoint with regard to the specification and verification of collaborative behavior features of component-based systems. Another pertinent issue regarding model V&V with reuse is the inclusion of built-in-test and interface violation detector mechanisms [Edwards 2002] with model components to facilitate ease of reuse and increase in confidence. This is due to the fact that developers need to test and verify reused models in their new context to systematically determine if they satisfy the constraints imposed by the context.

4.3 Conceptual Model Consistency

Rapid, distributed, and collaborative model development along with perpetual and complex change characterizes today's development of software-intensive mission critical simulation systems. The costs associated with detecting and fixing errors in such software-intensive simulations can amount to a significant percentage of overall development cost. The distributed and collaborative engineering of simulation model conceptualizations and specifications introduces consistency and coherence problems. Furthermore, fixing errors late in the life cycle is more costly due to the overall impact of code level changes. Hence, error reduction earlier in the simulation modeling lifecycle is needed. Model-based verification methodologies that could detect inconsistencies and behavioral defects at the conceptualization and model specification

stage is likely to alleviate these problems. The selective application of formal methodologies and the focus on error identification and correction distinguishes model-based verification approach from more traditional formal methods approaches with deductive apparatus. In particular, formal consistency analysis and management of UML structural analysis and design models and analytical model checking of UML behavioral design models based on their finite state reachability graphs are open issues.

4.4 Emergent Issues in Agent-Based Simulation Model V&V

Agent-based modeling and multi-agent simulations are promising emerging areas in simulation modeling. They represent an exciting new synthesis of AI, software engineering, and simulation modeling to simulate complex adaptive systems that constitute autonomous, intelligent, reactive, proactive, and goal-directed entities. Agent theory introduces new abstraction, organization, and decomposition mechanisms for system simulation. The emphasis on collaboration, coordination, task distribution, and adaptive behavior requires agent-based simulations to deploy specific algorithms and methods. Existing modeling paradigms do not scale and accurately characterize more complex, real-world systems (i.e., biological, organizational) that continuously change, adapt, and evolve to fit into their context. New paradigms arising from complex adaptive systems as well as artificial life theories are attracting the attention of model developers. V&V of distributed multi-agents systems that incorporate complexity through interactions, patterns of adaptive organizations, and processes requires new means to analyze and assess the accuracy of their emergent, mostly unpredictable behavior. The pervading notions of causality, mechanism, non-linear dynamics, process, and flows need to be addressed by new V&V methodologies.

4.5 Performance Testing of Distributed Active Systems with Publish / Subscribe Architectures: HLA Performance Benchmarking

As agencies shift and integrate their essential simulation services onto distributed software intensive message-oriented middleware platforms (C4I), failures as well as performance bottlenecks would result in unacceptable and ineffectual simulations. Configurable performance benchmarking environments would be valuable for large-scale Publish/Subscribe (P/S)-based distributed simulation infrastructures such as High Level Architecture (HLA) mandated by Defense Modeling and Simulation Office. To the best of our knowledge, no such architecturally proper HLA performance benchmarking tools yet exists. Hence, a new test methodology and supporting environment for emergent distributed active systems that deploy P/S architectures are needed. In such simulation architectures information must be disseminated from many federate and C4I node publishers to large number of subscribers, while subscribers must have the ability to select and extract the data of interest out of a dynamically changing information. Knowing how well a distributed messaging infrastructure performs under heavy load with a large number of concurrently connected federates is a key factor in planning a deployment strategy. Numerous companies and academic centers are developing and transitioning commercial P/S type of message-oriented middleware. However, the science of testing messaging middleware and frameworks is an uncharted territory as there are no industry-accepted benchmarks. To meet this need, P/S benchmarking environments are needed to (1) set up and design formal P/S performance experiments that would facilitate semi-automatic generation of scenarios, (2) derive benchmark configuration specifications from the scenarios, (3) configure, execute, and

distributively monitor the benchmark execution, and (4) perform statistical/visual post-analysis consistent with the experiments designed by the verification engineer.

4.6 Safe Model Test Boundaries: Managing the Evolution of Models

It is a well-known fact that almost two-thirds of software-intensive simulation development time is devoted to maintenance and evolution. In particular, the current trend towards using open standards supported by COTS in developing open distributed federations of simulation systems increases the likelihood of constant system modernization and evolution. The usage of COTS components and the need for modernization (i.e., to achieve information superiority) and C4I integration introduce upgrade problems. That is, as new or upgraded models (i.e., sensor models) are inserted into the baselines, model developers spend enormous amount of time to instill back confidence in the legacy systems and models to assure backward compatibility. Although this problem is not a new challenge, it is becoming a significant issue, as the new generation of systems is more complex, distributed, and COTS-driven. Software engineering and simulation directorates in military as well as commercial organizations invest increasingly large amount of effort to keep pace with the impacts of new technology insertion or adaptive modifications. There is an urgent need for tools that would at least help manage instilling back confidence on the legacy models with regard to their compatibility to the upgraded or modified baseline systems. This would enable system engineers to focus on the essential high-risk and error-prone areas of the models and its interfaces, while the identified unimpacted components are recertified automatically without human intervention using existing test suites and drivers.

4.7 Synthetic Natural Environment V&V

Driven by fiscal constraints, increasing pressure exists to employ realistic training simulations for driving exploration into new, more effective methods for modeling activities. The synthetic natural environment (SNE) models, upon which operational system models impact, constitute the environmental ground truth. The environmental ground truth contains both the data (i.e., terrain, atmosphere, ocean, space) and the models that represent the internal dynamics that effect and impact the environmental data. Recent DMSO initiatives such as *Integrated Natural Environment Authoritative Representation Process* require existing environmental data and models to be retrieved based on user-formulated constraints. The retrieved data that satisfies the user requirements need to be composed to derive complete synthetic environments. The shorter timescales required as well as the need to rapidly incorporate such SNE components demanded by training applications bring similar challenges. In particular, cost reduction in SNE development using parameterized scenario generation with compositional modeling is the core problem that needs to be addressed. Component-based development of complex SNE components requires domain-specific repository organization with intelligent component retrieval mechanisms along with reasoning methods for *conflict identification* before component assembly [Yilmaz *et al.* 2002]. Furthermore, merging disparate SNE components needs sophisticated data schema V&V, interoperability analysis, diagnosis, and merge tools. The major programs that will be supported by this research are: (1) advance concept development and defense planning, (2) R&D, acquisition, and military doctrine development, and (3) training, exercise, and military operations domains. Other application domains that would enjoy the advancements in rapid generation and certification of synthetic environments are the GIS and

C4I fields. These fields require the use of 2D and 3D data and models for visualization, analysis, and evaluation in support of decision-making tasks.

5. Recommendations

Model V&V is a confidence building activity during which the acceptability and credibility of a simulation model is established. The increasingly complex software-intensive nature of simulation models as well as their integration with real and virtual application domains require us to focus on quality in addition to accuracy. Product quality attributes such as maintainability, performance, and testability need to become essential indirect measures for acceptability and credibility of simulations. Furthermore, the increasing complexity is requiring managers to incorporate explicit risk management and SQA activities interleaved with the development process. Incorporating risk management into the simulation model development life cycle will also provide a compelling business case for comprehensive, systematic V&V. This is due to the fact that the avoided costs associated with business/mission risks as a result of V&V will be evident. The different opinions on management of V&V is also an important concern. Independent V&V is urged for successful simulation and software development studies [Lewis 1992]. For instance, as Arthur and Nance [2000] argue, lack of independent V&V in simulation modeling community is a recipe for failure. The need for achieving *technical*, *financial*, and *managerial* independence to establish the credibility of simulation models is becoming more and more apparent.

Building models from available COTS components is strongly advocated by academicians. The practitioners and managers support and are consent with the claims that compositional component-based modeling can indeed positively effect the development cost and effort as well as quality of models. However, the current state of the practice with regard to the modeling with and for reuse is still not mature. More research is needed to establish the principles of predictable compositional modeling.

The competency of V&V staff is another issue of concern that requires attention. Continuous rapid change in simulation development paradigms and methodologies makes some widely adopted V&V methods obsolete. A career path for V&V practitioners that constitutes continuous training and learning is essential.

6. Conclusions

This paper has presented an overview of verification and validation concepts and trends in the areas of modeling and simulation in particular and software development in general. Rapid advances in the use of sophisticated models as well as in the tools, techniques, and methods used to produce them produce both challenges and opportunities. These will create a continuing need for new innovative V&V technologies and strategies in the future.

References

- Addy, E. (1997). The Application of V&V within Reuse-Based Software Engineering. *Proceedings of the Eighth Annual Workshop on Software Reuse*.
- America, P. (1989). Issues in the Design of a Parallel Object-Oriented Language. *Formal Aspects of Computing*. 1, 4, pp. 366-411.
- Aoyama, M. (1998). Agile Software Process and Its Experience. *Proceedings of the 20th International Conference on Software Engineering*. pp. 3-12.
- Arthur, J. D. and Nance, R.E. (2000), Verification and Validation without Independence: A Recipe for Failure. *Proceedings of the 2000 Winter Simulation Conference* pp. 859-865.
- Balci, O. (1994). "Principles of Simulation Model Validation, Verification, and Testing." Technical Report ncstrl.vatech_cs//TR-94-24, Computer Science, Virginia Polytechnic Institute and State University.
- Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison Wesley Longman.
- Berard, E. (1992). *Essays on Object-Oriented Software Engineering*, Volume 1. Englewood Cliffs, NJ: Prentice-Hall.
- Binder, R. (1994). "Testing Object-Oriented Systems: A Status Report," *American Programmer* 7,4 (April). pp. 22-28.
- Binder, R. (1995). "Trends in Testing Object-Oriented Software." *IEEE Computer*, October 1995. pp. 68-69.
- Binder, R.V. (2000), *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Reading, MA: Addison-Wesley.
- Boehm, B. W. (1984), Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, Jan., pp. 75-88.
- Boehm, B. W. (1988). A Spiral Model for Software Development and Enhancement. *IEEE Computer*. 21, 5, pp. 61-72.
- Boehm, B. (2002). Get Ready for Agile Methods, With Care. *IEEE Computer*. 35, 1, pp. 64-69.
- Booch, G. (1991). *Object-Oriented Design with Applications*. Menlo Park, CA: Benjamin Cummings.
- Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin Cummings.
- Brooks, P. F. (1995). *The Mythical Man-Month (Anniversary Edition)*. New York: Addison-Wesley.
- Cheatham, J. T. and Mellinger, L. (1990). Testing Object-Oriented Software Systems. *Proceedings of the 1990 ACM 18th Annual Computer Science Conference*. pp. 161-165.
- Chen, Y. T. and Low, C. K. (1995). Dynamic Data Flow Analysis for C++. *Proceedings of the Asia-Pacific Software Engineering Conference*. December 1995. pp. 22-28.
- Chung, C.-M. and Lee, M.-C. (1992). Object-Oriented Programming Testing Methodology. *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*. June 1992. pp. 378-385.

- Clarke, E. M., Grumberg, O., and Long, D. E (1994). Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16, 5, pp. 1512-1542.
- Cramer J., Doberkat, E., and Goedicke, M. (1994). Formal Methods. *Software Reusability*. Schafer *et al.* (Eds). Chichester, England: Ellis Horwood.
- DeGrace, P. and Stahl, L. H. (1990). *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*. Upper Saddle River, NJ: Prentice Hall.
- Dery R., Landry, M., and Banville, C (1993). Revisiting the Issue of Model Validation in OR: An Epistemological view. *European Journal of Operations Research*. 66, 2, pp. 168-184.
- Deutsch, M. S. (1982). *Software Verification and Validation*. Englewood Cliffs, NJ: Prentice Hall.
- Dijkstra, W. E. (1976). *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall.
- Doong, R.-K. and Frank, P. (1994). The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Transactions on Software Engineering and Methodology*. 3, 4 (April), pp. 101-130.
- D'Souza, J. R. and LeBlanc, R. J. (1994). Class Testing by Examining Pointers. *Journal of Object-Oriented Programming*. 7, 4, pp. 33-39.
- Edwards, S. (2002). Making the Case for Assertion Checking Wrappers. *Proceedings of the RESOLVE Workshop 2002*. pp. 28-42. Ohio-State University, Columbus, Ohio.
- Ernst, G. W., Navlakha, J. K., and Ogden, W. F. (1982). Verification of Programs with Procedure Type Parameters. *Acta Inf.* 4, 2, pp. 522-543.
- Fiadeiro, J. L. and Maibaum, T. (1994). Verifying for Reuse: Foundations of Object-Oriented System Verification. *Proceedings of Theory and Formal Methods of Computing*. 94, pp. 235-257.
- Fiedler, P. S. (1989). Object-Oriented Unit Testing. *Hewlett-Packard Journal*, 40, 2, 69-75.
- Firesmith, G. D. (1993). Testing Object-Oriented Software. *Proceedings of the 11th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '93)*. Englewood Cliffs, NJ: Prentice-Hall. pp. 407-426.
- Floyd, W. R. (1968). Assigning Meanings to Programs. *Proceedings of Applied Mathematics*. American Mathematical Society. 19, pp. 19-32.
- Gannon, J., McMullin, P., Hamlet, R. (1981). Data Abstraction Implementation, Specification, and Testing. *ACM Transactions on Programming Languages and Systems*. 3, 3 (July 1981), pp. 211-223.
- Gluch, P. D. and Weinstock, C. B. (1998). Model-based verification: A Technology for Dependable System Upgrade. *CMU/SEI - 98-Technical Report-009*.
- Goguen, J. A. (1986). Reusing and Interconnecting Software Components. *IEEE Computer*. (February 1986), pp. 16-27.
- Goguen, J. A. and Meseguer, J. (1986). Extensions and foundations of object-oriented programming. *SIGPLAN Notices*. 10, 21, pp. 153-162.
- Graham, A. J., Drakeford, A. C. T., and Turner, C. D. (1993). The Verification, Validation, and Testing of Object-Oriented Systems. *BT Technology Journal*. 11, 3, pp. 79-88.

- Guaspari D., Marceau C., and Polak W. (1990). Formal Verification of Ada Programs. *IEEE Transactions of Software Engineering*. 16, 9, pp. 1058-1075.
- Harrold, J. M., McGregor, J. D., and Fitzpatrick, K. J. (1992). Incremental Testing of Object-Oriented Class Structures. *Proceedings, of the 14th International Conference on Software Engineering*. pp. 68-80.
- Hayes, H. J. (1994). Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach. *Proceedings of Object-Oriented Methodologies and Systems*, E. Bertino, S. Urban (Eds.), LNCS 858. Berlin: Springer-Verlag. pp. 205-220.
- Hegazy, W. A. (1989). The Requirements of Testing a Class of Reusable Software Modules. *Ph.D. dissertation*. Department of Computer and Information Science, Ohio State University, Columbus, OH.
- Hoare, A. C. (1969). An Axiomatic Basis for Programming. *Communications of the ACM*. 12, 10 (October 1969).
- Hughes, M. and Stotts, D. (1996). Daistish: Systematic Algebraic Testing for OO Programs. *Software Engineering Notes*. 21, 3, pp. 53-61.
- IEEE/EIA 12207.1-1997*. (1998). Software Life Cycle Processes - Life Cycle Data.
- Jacobson, I. (1992). *Object-Oriented Software Engineering: A Use-Case Driven Approach*. New York: Addison Wesley.
- Jain, R. (1991). *The Art of Computer Systems Performance Analysis*. New York: Addison-Wesley.
- Jorgensen, C. P. and Erickson, C. (1994). Object-Oriented Integration Testing. *Communications of the ACM*. 37, 9, pp. 30-38.
- Kandelin, A. N. and O'Leary, D. E. (1995). Verification of Object-Oriented Systems: Domain Dependent and Domain-Independent Approaches. *Journal of Systems and Software*. 29, pp. 261-269.
- Krone, J. (1988). The Role of Verification in Software Reusability. *Ph.D. dissertation*. Department of Computer and Information Science, Ohio State University, Columbus, OH.
- Kung, D. J., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. (1993). Design Recovery for Software Testing of Object-Oriented Programs. *Proceedings of the Working Conference on Reverse Engineering*. Los Alamitos, CA: IEEE Computer Society Press. pp. 202-211.
- Kung, D. J., Suchak, N., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. (1994). On Object State Testing. *Proceedings of the 18th International Computer Software & Applications Conference*. pp. 222-227.
- Kung, D. J., Hsia, P., Toyoshima, Y., Chen, C., Kim, Y., and Song, Y. (1995). Developing an Object-Oriented Software Testing and Maintenance Environment. *Communications of the ACM*. 38, 10, pp. 75-86.
- Landry, M., Malouin, J. L, and Oral, M. (1983). Model Validation in Operations Research. *European Journal of Operations Research*. 14, 3, pp. 207-220.
- Law, A. M. and Kelton, W. D. (1991). *Simulation Modeling and Analysis (2nd ed)*. New York: McGraw-Hill.

- Lewis, R. O. (1992). *Independent Verification & Validation: A Life Cycle Engineering Process for Quality Software*. New York: Wiley.
- Lindland, I. O., Sindre, G., and Solvberg, A. (1994). Understanding Quality in Conceptual Modeling. *IEEE Software*. 11, 2, pp. 42-49.
- Luckham, D., von Henke, F., Krieg-Bruckner, B., and Owe, O. (1987). *ANNA: A Language for Annotating Ada Programs*. Berlin: Springer-Verlag.
- Martin, J. and Odell, J. J. (1992). *Object-Oriented Analysis & Design*. Englewood Cliffs, NJ: Prentice-Hall.
- Maurer, F. and Martel, S. (2002). Extreme Programming: Rapid Development for Web-Based Applications. *IEEE Internet Computing*. Jan./Feb. 2002. pp. 86-90.
- McGregor, D. J. and Dyer, D. M. (1993). A Note on Inheritance and State Machines. *Software Engineering Notes*. 18, 4, pp. 61-69.
- McGregor, D. J. and Korson, T. (1994). Integrating Object-Oriented Testing and Development Processes. *Communications of the ACM*. 37, 9, pp. 59-77.
- Myers, J. G. (1979). *The Art of Software Testing*. New York: Wiley-Interscience.
- Meyer, B. (1988). *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice-Hall.
- Nance, R. E. (1987). The Conical Methodology: A Framework for Simulation Model Development. *Methodology and Validation*. San Diego, CA: SCS. pp. 38-43.
- Naylor, T. H. and Finger, J. M. (1967). Verification of Computer Simulation Models. *Management Science*. 14, 2, pp. 92-101.
- Overstreet, C. M. and Nance, R. E. (1985). A Specification Language to Assist in Analysis of Discrete Event Simulation Models. *Communications of the ACM*. 28, 2, pp. 190-201.
- Pace, D. K. (1999). Development and Documentation of a Simulation Conceptual Model. *Proceedings of the Fall 1999 Simulation Interoperability Workshop*. March 15-19, 1999.
- Parrish, A., Borie, R. B., and Cordes, D. W. (1993). Automated Flow Graph-Based Testing of Object-Oriented Software Modules. *Journal of Systems and Software*. 23, 2, pp. 95-109.
- Paulk, M. C. (2001). Extreme Programming from a CMM Perspective. *IEEE Software*. 18, 6, pp. 19-26.
- Perry, E. D. and Kaiser, G. E. (1990). Adequate Testing and Object-Oriented Programming. *Journal of Object-Oriented Programming*. 2, 5 (January/February), pp. 13-19.
- Poston, M. R. (1994). Automated Testing from Object Models. *Communications of the ACM*. 37, 9, pp. 48-58.
- Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach (5th ed.)*. Boston, MA: McGraw Hill.
- Purchase, A. J. and Winder, R. L. (1991). Debugging Tools for Object-Oriented Programming. *Journal of Object-Oriented Programming*. 4, 3 (June 1991), pp. 10-27.
- Rakitin, S. R. (2001). *Software Verification and Validation for Practitioners and Managers (2nd ed)*. Boston: Artech House.

- Rumbaugh J., Blaha, M, Premerlani, W., Eddy F., and Lorenzen, W. (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall.
- Sargent G. R. (1996). "Verifying and Validating Simulation Models," In *Proceedings of the 1996 Winter Simulation Conference*, pp. 55-64.
- Sargent, R.G. (1999), "Validation and Verification of Simulation Models," In *Proceedings of the 1999 Winter Simulation Conference*, P.A. Farrington, H.B. Nembhard, D.T. Sturrock, and G.W. Evans, Eds., IEEE, Piscataway, NJ, pp. 39-48.
- Schulmeyer, G. G. and MacKenzie, G. R. (2000). *Verification & Validation of Modern Software-Intensive Systems*. Upper Saddle River, NJ: Prentice Hall.
- Shaw, M., ed. (1981). *ALPHARD: Form and Content*. New York: Springer-Verlag.
- Smith, D. M. and Robson, D. J. (1990). Object-Oriented Programming: The Problems of Validation. *Proceedings of the 6th International Conference on Software Maintenance*. Los Alamitos, CA: IEEE Computer Society Press. pp. 272-282.
- Standish Group (1995). *Chaos*. Standish Research Paper, www.standishgroup.com/chaos.html.
- Subramanian, S., Tsai, W. T, and Kirani, S. H. (1994). Hierarchical Data Flow Analysis for O-O programs. *Journal of Object-Oriented Programming*. May 1994, pp. 36-46.
- Turner, D. C. and Robson, D. J. (1993). The State-Based Testing of Object-Oriented Programs. *Proceedings of the International Conference on Software Maintenance*. September 1993, Los Alamitos, CA: IEEE Computer Society Press. pp. 302-310.
- Weide, W. B. and Hollingsworth, J. E. (1992). On Local Certifiability of Software Components. *Proceedings of the 5th Annual Workshop on Software Reuse*.
- Wing, M. J. and Farahani, M. V. (1995). Model Checking Software Systems: A Case Study. *SIGSOFT Software Engineering Notes*. 20, 4, pp. 128-39.
- Wirfs-Brock R., Wilkerson, B., and Wiener, L. (1990). *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall.
- Wohlin C. and P. Runeson (1994). Certification of Software Components. *IEEE Transactions on Software Engineering*. 20, 6, pp. 494-499.
- Yilmaz, L. and Balci, O. (1997). Object-Oriented Simulation Model Verification and Validation. *Proceedings of the 1997 Summer Computer Simulation Conference*. July 13-17, 1997. The Society for Computer Simulation International.
- Yilmaz, L. (1998). Verification and Validation Guidelines for Object-Oriented Simulation Models. *Proceedings of the 1998 Summer Computer Simulation Conference*. July 19-22, 1998. The Society for Computer Simulation International. pp. 645-646.
- Yilmaz, L. (2000). Issues in the Certification of Reusable Object-Oriented Model Components: The Coordination Perspective. *Proceedings of the 2000 International Conference on Web-Based Simulation and Modeling*. pp. 55-61.
- Yilmaz, L. (2001). Automated Object-Flow Testing of Dynamic Process Interaction Models. *Proceedings of the 2001 Winter Simulation Conference*.
- Yilmaz, L., Nguyen, T., DeLaCruz, J., and Kohler, T. (2002). Distributed Repositories of Reasonable Ontologies for SNE Component Retrieval and Reuse. *Proceedings of the*

Interservice/Industry Training, Simulation, and Education Conference (IITSEC'02). (to appear).

Yilmaz, L. and Edwards, S. (2002). Specifying and Verifying Collaborative Behavior in Component-Based Systems. *RESOLVE Workshop 2002*. pp. 95-104.