Specification Languages: Understanding Their Role in Simulation Model Development

C. Michael Overstreet
Richard E. Nance
Osman Balci
Lynne F. Barger

TR 87-7

Technical Report TR-87-7†

Specification Languages: Understanding Their Role in Simulation Model Development*

C. Michael Overstreet‡
Richard E. Nance±
Osman Balci§
Lynne F. Barger§

December 1986

* This research was sponsored in part by the Naval Sea Systems Command and the Office of Naval Research under Contract N60921-83-G-A165 through the Systems Research Center at Virginia Tech.

‡ Department of Computer Science, Old Dominion University.

± Department of Computer Science, and Systems Research Center, Virginia Tech.

§ Department of Computer Science, Virginia Tech.

† Cross-referenced as SRC-87-001, Systems Research Center, Virginia Tech.

Technical Report SRC-87-001†

Specification Languages: Understanding Their Role in Simulation Model Development*

C. Michael Overstreet‡
Richard E. Nance±
Osman Balci§
Lynne F. Barger§

December 1986

* This research was sponsored in part by the Naval Sea Systems Command and the Office of Naval Research under Contract N60921-83-G-A165 through the Systems Research Center at Virginia Tech.

‡ Department of Computer Science, Old Dominion University.

± Systems Research Center and Department of Computer Science, Virginia Tech.

§ Department of Computer Science, Virginia Tech.

† Cross-referenced as TR-87-7, Department of Computer Science, Virginia Tech.

ABSTRACT

Current software specification techniques and specification languages are reviewed, emphasizing research activities in software specification languages. Alternate software life cycle models are described and compared to a simulation life cycle model. The importance of constructing a model specification before creating a programmed model is emphasized. Disadvantages in using simulation programming languages as model specification languages are discussed. The multiple uses which are made of a model specification are presented; these uses correspond to the alternate uses made of a requirements specification for general software. To evaluate where specification tools for general software will be effective for simulation modeling, both areas where the simulation life cycle corresponds to a general software life cycle and areas in which they differ are characterized. Important conclusions are that: (1) specification methodologies are highly dependent on the phase of the software life cycle and the use to be made of the specification, (2) both general software and simulation communities lack a clear perception of the proper form for specifications, and (3) evaluation of proposed methodologies and tools is inadequate.

CR Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications — *Languages*; I.6.m [**Simulation and Modeling**]: Miscellaneous

General Terms: Languages

Additional Keywords and Phrases: Model life cycle, simulation model specification, software life-cycle models

1. INTRODUCTION

Current simulation programming languages (SPLs) are rooted in the early 1960s, the very active period in which the most popular languages emerged: GPSS, SIMSCRIPT, CSL, and SIMULA. Even a language appearing later, such as SLAM, has clear ancestral links to a predecessor developed during this time period (GASP). Despite the evolutionary changes in all of the SPLs, certain genetic traits remain dominant and unfortunately limiting in the development of more extensive, complex models. One such trait is the continuing emphasis on *model execution* to the detriment of *model specification.*

This criticism must be interpreted in the proper context. No doubt, a decision to acquire an SPL is likely to be dominated by execution-related factors, such as:

(1) comparative model execution times,

(2) programmer knowledge of an underlying host language, such as Fortran for SLAM or Algol for SIMULA,

(3) report generation formatting flexibility, or

(4) "neat" output animation conveniently employed.

Even more influential is the tradeoff in investment versus return for adding "bells" and "whistles," such as color graphics and iconic output displays, rather than addressing the fundamental issues embodied in how a modeler translates undefined conceptual views into defined communicative forms. This paper addresses that issue by drawing from the experiences and findings of the software engineering community with regard to *program specification* to better understand the challenge of *model specification.*

1.1. The Meaning of Specification

The importance of specification, *i.e.* describing *what* behavior is to be produced without the details of *how* that behavior is produced, is clearly recognized in the software engineering community. Significant concepts, such as data abstraction, pertain to specification rather than to implementation, for as Liskov notes (Liskov 1984, p. 56):

A data abstraction is an idea of a kind of behavior, and the description of that behavior is not given in a programming language. It should be given in a specification language. A programming language is just a tool for implementing that behavior once you know what it is.

The desirable characteristics of a good specification tool (language) are not the same as those of a good implementation tool (language). Although this realization has been affirmed in selected large simulation modeling projects (see (Pohoski 1981) for example), the principle of separation of model execution from model specification has yet to be widely accepted.

1.2. An Approach to Simulation Model Specification

The function of specification in software development is explained in the software life cycle model. Consequently, we introduce the software and model life cycle views in Section 2 to understand the similarities and differences. Section 2 concludes with a comparison of the specification roles in the life cycle models. The understanding of specification is sharpened and focused by a review of software specification languages in Section 3, which concludes with a taxonomy. Against the general software backdrop, the prior work in simulation model specification is reviewed in Section 4. Section 5 draws the software and model specification ideas together in an evaluative characterization of the needs for simulation modeling. A summary with conclusions in Section 6 completes the paper.

2. LIFE-CYCLE MODELS

In this section the software life-cycle models and the simulation model life cycle are described and the roles of specification in software and simulation model life cycles are compared.

2.1. The Software Life Cycles

The seven major software life-cycle models proposed in the literature are briefly described below in chronological order.

2.1.1. The Stagewise Model

One of the earliest software life-cycle models is the stagewise model given in (Bennington 1956). This model proposes nine successive stages for software development: (1) operational plan, (2) machine and operational specifications, (3) program specifications, (4) coding specifications, (5) coding, (6) parameter testing (specifications), (7) assembly testing (specifications), (8) shakedown, and (9) system evaluation.

2.1.2. The Waterfall Model

This model recommends that software be developed as a sequence of tasks "waterfalling" into one another as characterized by Boehm in Figure 1 (Boehm 1976).

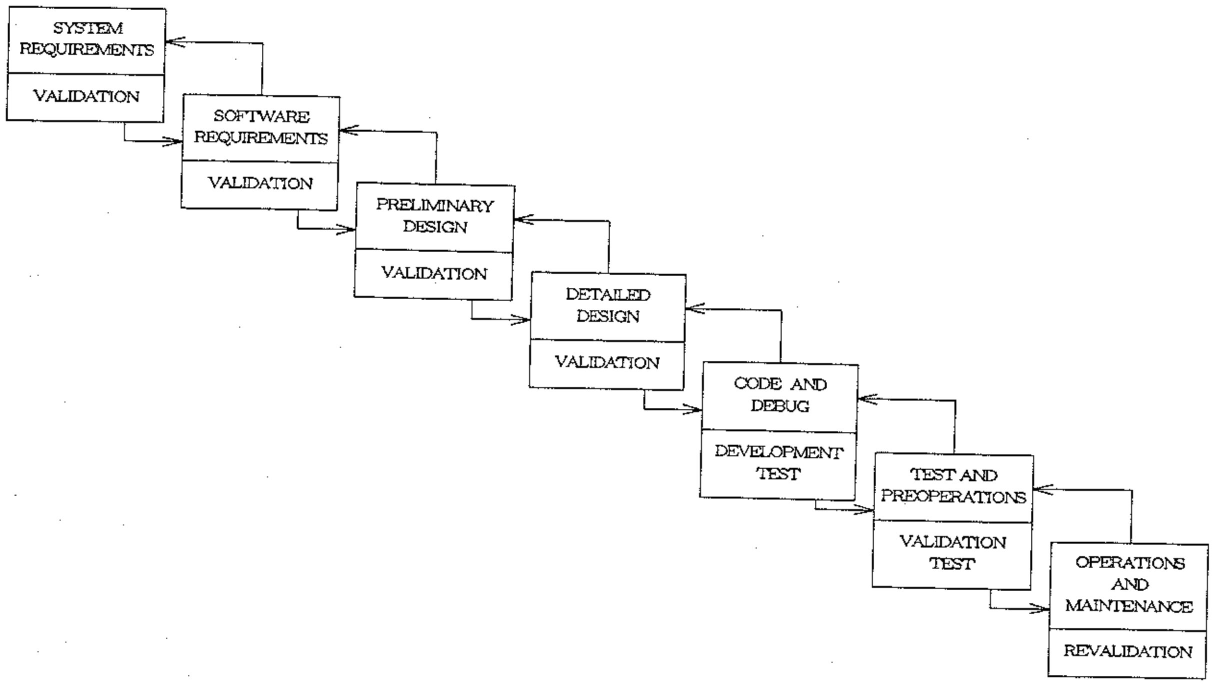

Figure 1. Boehm's Waterfall Model

The original treatment of the waterfall model is given in (Royce 1970). Boehm expands each step to include a validation and verification activity to cover high-risk elements, reuse considerations, and prototyping (Boehm 1976). Distaso further elaborates on the waterfall model by including such practices as incremental development and top-down integration (Distaso 1980).

2.1.3. The Automation-Based Paradigm

Balzer et al. (1983) propose a life-cycle model, shown in Figure 2, for incorporating the capabilities of automatic programming, program transformation, and "knowledge-based software assistant."

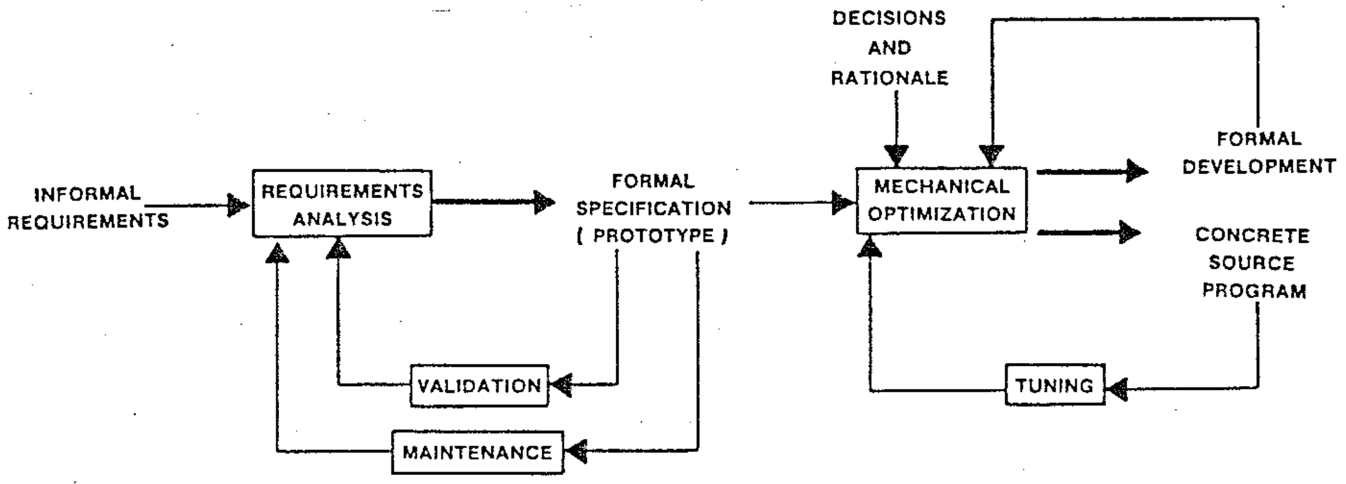

Figure 2. The Automation-Based Paradigm.

The technology needed to support this paradigm does not yet exist. However, the benefits to be gained are so significant that, if achieved, it could profoundly change the way the software is developed.

2.1.4. Mixed Models

Recognizing that software is not homogeneous, Lehman (1980) categorizes programs into three classes, S, P, and E. S-programs constitute the class of software whose function is formally defined by and derivable from a *specification*. Examples of this class include numerical algorithms, function evaluation in a specified domain, lowest common multiple of two integers. P-programs are models constructed for real-world *problem* solution. Both the problem statement and its solution model (approximate) the real-world situation; for example, a program to play chess, a model for weather prediction, a model of the U.S. economy. An E-program is the one that mechanizes a human or societal activity by being *embedded* in the real-world it models. The existence of the program alters the application environment, significantly changing the original problem. Examples of E-programs abound: transaction-related business systems, office automation systems, air-traffic control systems.

Similar to Lehman's categorization, Giddings (1984) classifies software as domain independent (corresponding to S-programs), domain dependent—experimental (corresponding to P-programs), and domain dependent—embedded (corresponding to E-programs).

For domain dependent software (P and E), Giddings (1984) proposes a life cycle, shown in Figure 3, that accommodates evolutionary development through prototyping.

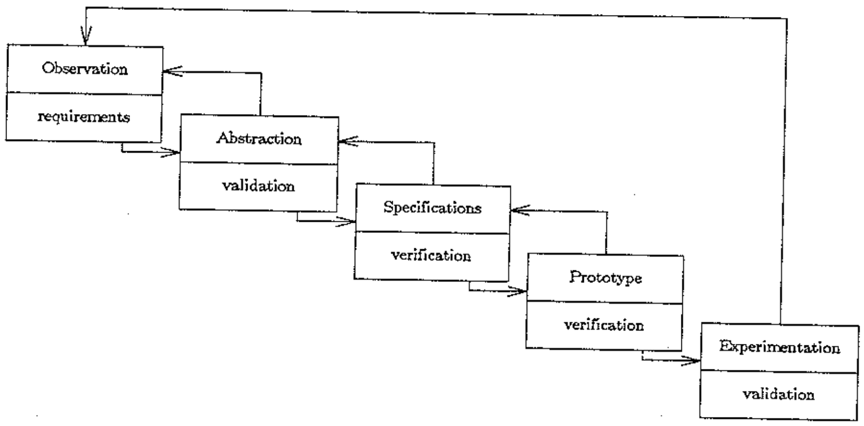

Figure 3. Giddings' Domain Dependent Software Life Cycle.

The process of building successive prototypes through iterative refinement and experimentation continues until an acceptable prototype is achieved.

2.1.5. The Two-Leg Model

This model, shown in Figure 4, is depicted as two legs in the form of an inverted V (Lehman 1984).

The left leg is called abstraction, and it represents a transformation from an application concept to a formal specification. The right leg is called reification, and it represents a transformation from the formal specification to an operational system. Lehman et al. (1984) developed a model for the reification process in terms of decomposition, backtrack, and recursion.

2.1.6. The Operational Approach

The operational approach, shown in Figure 5, is based on separation of problem-oriented from implementation-oriented concerns, and all its features can be derived from that philosophy (Zave 1984a). The operational specification, which is only problem-oriented, is a complete and executable representation of the proposed system. During the transformation phase, the specification is

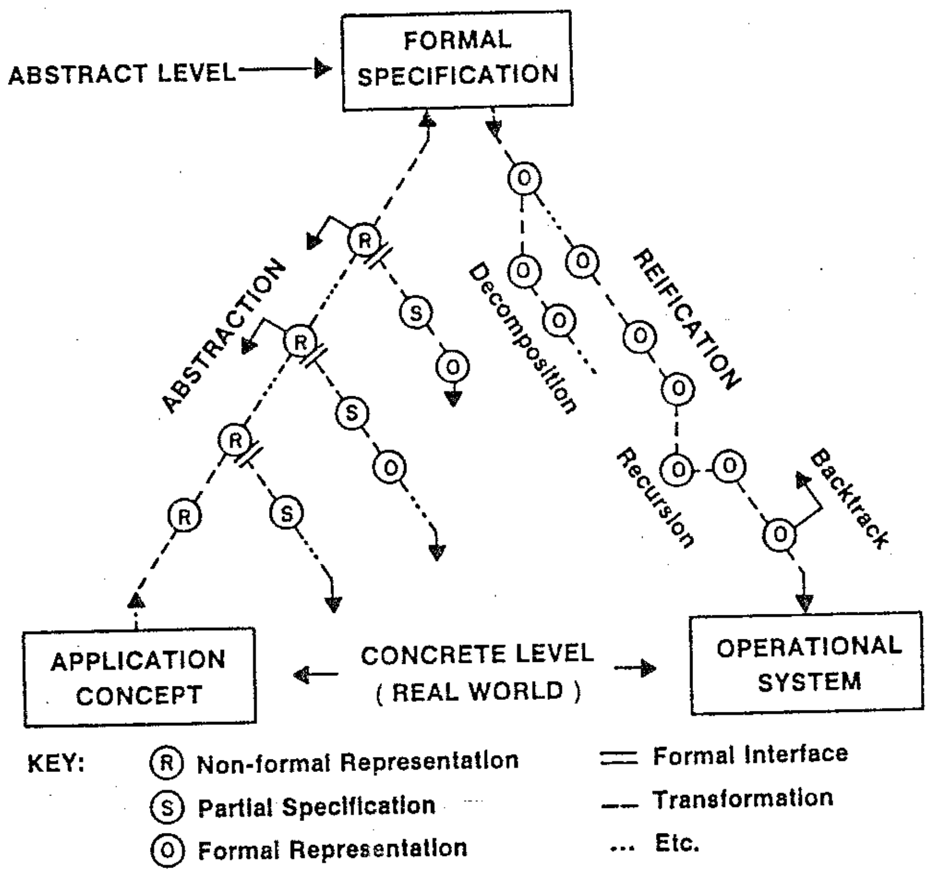

Figure 4. The Two-Leg Model.

Problem Understanding → *Specification* → Operational Specification → *Transformation* → Transformed Specification → *Realization* → Solution System

Figure 5. The Operational Approach Life-Cycle Model.

subjected to a series of automated transformations that preserve external behavior but change the mechanisms producing the behavior so as to create an implementation-oriented (transformed) specification.

2.1.7. The Spiral Model

Based on experience with various refinements of the Waterfall Model as applied to large government projects, Boehm (1986) proposes the Spiral Model illustrated in Figure 6. This model provides

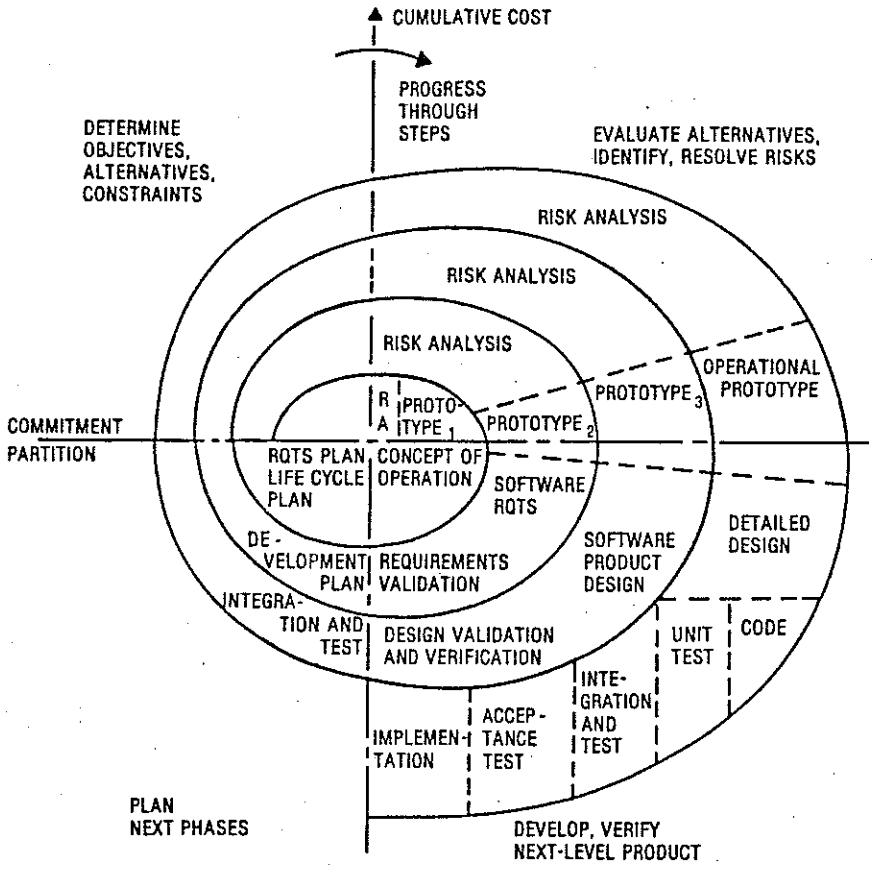

Figure 6. The Spiral Model.

a risk-driven approach and accommodates any appropriate mixture of specification-oriented, prototype-oriented, simulation-oriented, automatic transformation-oriented approaches to software development. In Figure 6, the radial dimension represents the cumulative cost incurred in accomplishing the steps to date; the angular dimension represents the progress made in completing each cycle of the spiral.

2.2. The Simulation Model Life Cycle

The simulation model life cycle is defined by Nance and Balci (Nance 1981; Balci 1986) and is illustrated in Figure 7. The oval symbols in Figure 7 represent the phases; the dashed arrows describe the processes which relate the phases to each other. The solid arrows refer to the credibility assessment stages.

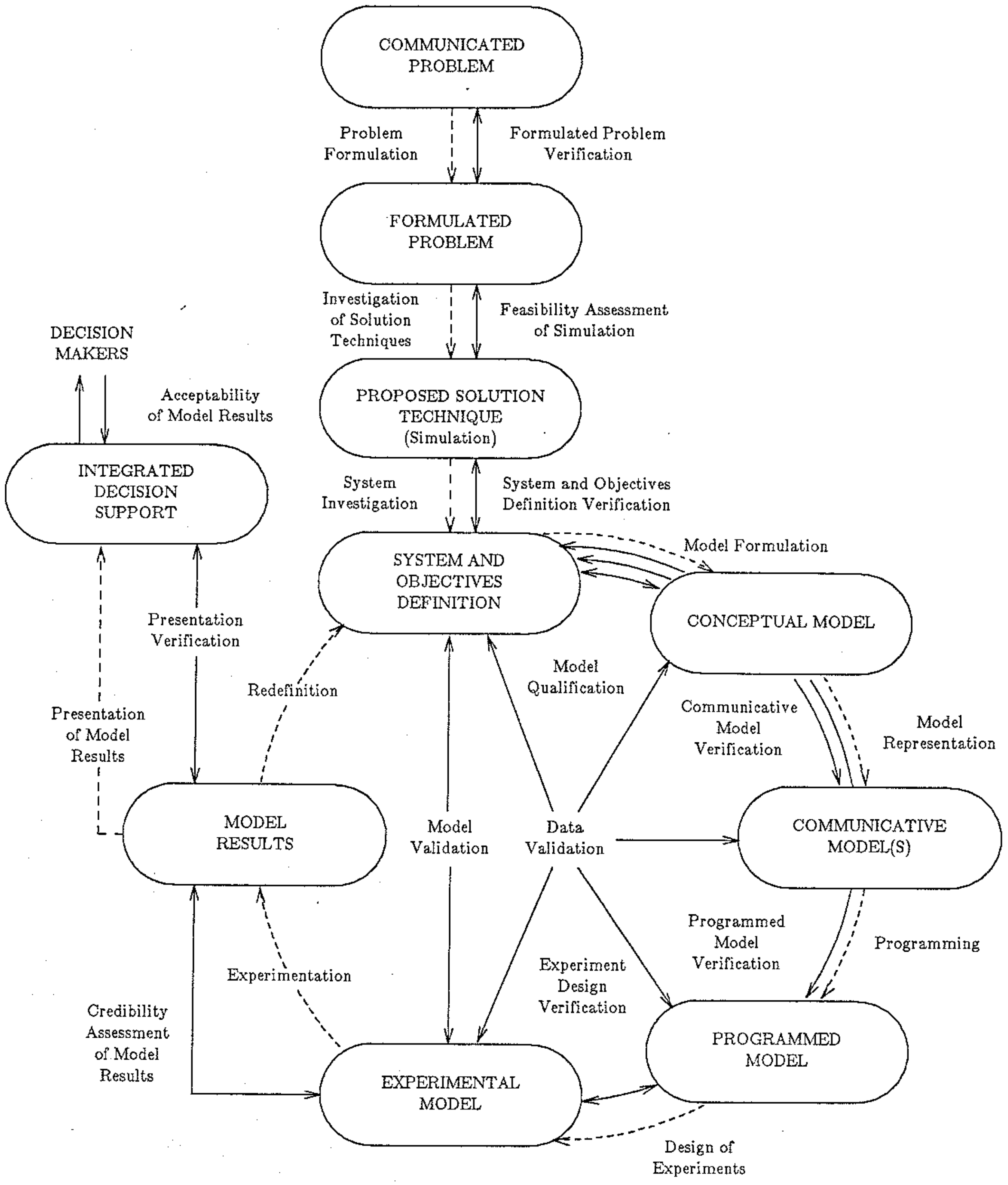
COMMUNICATED PROBLEM
Problem Formulation
Formulated Problem Verification
FORMULATED PROBLEM
Investigation of Solution Techniques
Feasibility Assessment of Simulation
DECISION MAKERS
Acceptability of Model Results
PROPOSED SOLUTION TECHNIQUE (Simulation)
INTEGRATED DECISION SUPPORT
System Investigation
System and Objectives Definition Verification
SYSTEM AND OBJECTIVES DEFINITION
Model Formulation
CONCEPTUAL MODEL
Presentation Verification
Model Qualification
Redefinition
Presentation of Model Results
Communicative Model Verification
Model Representation
MODEL RESULTS
Model Validation
Data Validation
COMMUNICATIVE MODEL(S)
Programmed Model Verification
Programming
Experimentation
Experiment Design Verification
Credibility Assessment of Model Results
PROGRAMMED MODEL
EXPERIMENTAL MODEL
Design of Experiments

Figure 7. The Simulation Model Life Cycle

The life cycle should not be interpreted as strictly sequential. The sequential representation of the dashed arrows is intended to show the direction of development throughout the life cycle. The life cycle is iterative in nature and reverse transitions are expected.

Since the advent of computers, specification has played a vital role in the development of software. Five of the nine stages of the Stagewise Model proposed by Bennington (1956) address specification.

Perhaps the most commonly used software life-cycle model is the Waterfall Model which incorporates four specification stages: system and software requirements specifications and preliminary and detailed design specifications. The importance of specification in the early stages of the software life cycle has stimulated the recognition of "software requirements engineering." Boehm (1976) defines this as "the discipline for developing a complete, consistent, unambiguous specification describing *what* the software product will do (but *not how* it will do it; this is to be done in the design specification)."

The formal specification that can be fully translated into an efficient implementation is the central notion of the automation-based paradigm which, if achieved, could profoundly change the way the software is developed. In this new paradigm, the specification serves as a prototype and the maintenance is performed on the specification rather than the implementation (source program). Thus, the maintenance problem is drastically simplified by directly modifying the specification closest to the user's conceptual model, which should be the least complex and most localized.

The Two-Leg Model describes the software life cycle as

$$\text{Application Concept (A)} \rightarrow \text{Formal Specification (S)} \rightarrow \text{Operational System (P)}$$

with iteration (i.e., P $\rightarrow$ A) where $\rightarrow$ represents a transformation process. The A $\rightarrow$ S transformation involves many iterations of "non-formal representation $\rightarrow$ partial specification $\rightarrow$ formal representation" until an acceptable S is achieved. The S $\rightarrow$ P reification process employs successive transformations of specifications using decomposition, recursion, and backtracking until P is obtained (i.e., $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \dots \rightarrow P$). The formal specification at the apex of the Two-Leg

Model plays the crucial "middle-man" role between A and P.

Two types of specification constitute the underpinnings of the Operational Approach: (1) operational specification — a problem-oriented, complete, and executable representation of the proposed system, (2) transformed specification — an implementation-oriented representation of the same system obtained through automated transformations from the operational specification.

The specifications within the domain dependent software life cycle are evolved through iterative refinement. Specification-oriented software development is one of the several approaches accommodated under the Spiral Model.

Specification plays an important role in the simulation model life cycle for the development of the communicative models. It is critical to detect errors as early as possible in the life cycle. Errors induced within the specification of the communicative model are either caught in much later phases resulting in a higher cost of correction or never detected resulting in the type II error — the error of accepting invalid results (Balci 1986).

3. THE SPECIFICATION TASK

The several software life-cycle models, although exhibiting numerous differences, demonstrate consensus regarding the need for *specifying* behavior prior to describing how that behavior is achieved. Both expressions — description of *what* behavior or *how* to realize that behavior — admit many forms. A specification language offers guidance in perceiving a system by the provision of concepts describing behavior, but more importantly, a specification language is the medium of communication for expressing this behavior.

In the attempt to understand "model specification," we review the many software specification languages found in software engineering. Typically, such a review can be organized around the phase of the life-cycle model supported by the language, for few support more than one. The categories of such an organization are illustrated in Table 1 for a generic life-cycle model (probably resembling revisions of Boehm's Waterfall Model (Boehm 1976) most closely). With understanding simulation model specification as the driving motivator, we find an augmented scheme to be preferable — one that includes the basic descriptive unit of a language: function, object, or process. Explanation of the "basic descriptive unit" and its use in classification follows.

Table 1. The Phases of a Generic Software Life-Cycle Model

REQUIREMENTS DEFINITION • Description of problem to be solved
FUNCTIONAL SPECIFICATION • Description of behavior of a system that solves problem • Emphasis on "What" system does — system is a black box • Expression of specifier's conceptual model of system • No algorithms or data structures given • Important communication link among designers, implementers, and customers
ARCHITECTURAL DESIGN • Identify modules to perform desired system behavior • Describe module effects and interfaces — module is a black box • Definition of internal system structure • Design emphasis shifts to "How"
DETAILED DESIGN • Describe algorithms and data structures needed to achieve desired behavior of each module • Specific instructions on *how* to code the system
IMPLEMENTATION • Translation of detailed design into an executable program • Testing program to see if it meets requirements
REFERENCES (Berzins 1985, Freeman 1983, Stoegerer 1984, Wasserman 1983, Yeh 1984, Zave 1984b)

3.1. A Classification of Software Specification Languages

The language classification, incorporating both the life-cycle phase and basic descriptive unit, is

Table 2. A Taxonomy of Specification Languages

BASIC DESCRIPTION UNIT	LANGUAGE	LIFE CYCLE PHASE	UNDERLYING MODEL	REFERENCES
FUNCTION	Algebraic Spec	F	Mathematical	(Gehani 1982) (Liskov 1975)
	AXES	F,A,D	Mathematical	(Hamilton 1976) (Hamilton 1983) (Martin 1985a) (Martin 1985b)
	Special	F	Mathematical	(Silverberg 1981) (Stoegerer 1984)
OBJECT	DDN	A	Message passing	(Riddle 1978a) (Riddle 1978b) (Riddle 1979) (Riddle 1980)
	MSG.84	F,A	Message passing	(Berzins 1985)
	PSL/PSA	F	ERA	(Stoegerer 1984) (Teichrow 1977) (Teichrow 1980) (Winters 1979)
	RDL	A,D	ERA	(Heacox 1979) (Stoegerer 1984)
	RML	F	ERA	(Borgida 1985) (O'Brien 1983)
	TAXIS	A,D	ERA	(Borgida 1985) (O'Brien 1983)
PROCESS	GIST	O	Stimulus response ERA	(Balzer 1980) (Balzer 1982) (Feather 1983) (Goldman 1980) (London 1982)
	PAISLey	O	Stimulus response	(Yeh 1984) (Zave 1979) (Zave 1982) (Zave 1984a) (Zave 1984b)
	RSL	F,O	Stimulus response	(Alford 1977) (Alford 1985) (Bell 1977) (Davis 1977) (Scheffer 1985) (Stoegerer 1984)

shown in Table 2. Each category is described briefly below.

3.1.1. Function Orientation

Function-oriented specification languages require that a system be defined as a series of functions which map inputs onto outputs. These languages enforce certain mathematical axioms thereby producing a *provably* correct specification (Martin 1985b). However, the mathematical basis of these languages creates several problems:

- The resulting specification is often unreadable and difficult to understand (Davis 1982).
- The languages are hard to learn and to use for people without strong mathematical backgrounds (Gehani 1982).
- The languages are applicable only for specifying small systems (Wasserman 1983). (AXES of Higher Order Software is an exception.)

3.1.2. Object Orientation

The object-oriented specification languages yield a model representation composed of object descriptions with each description containing the behavioral rules and characteristics for that object (Faught 1980). Some examples of object-oriented specification languages are listed in Table 2. Each of these languages assists in the production of object descriptions, but the exact form of the object description varies with the underlying model of the language. Two models commonly employed are the Entity-Relation-Attribute (ERA) model (Chen 1976) and the message-passing model (Goldberg 1984).

The ERA model (Stoegerer 1984) views a system as being composed of entities (objects), entity attributes, and relationships among entities. ERA-based languages are widely used for applications where static descriptions are necessary (such as databases), but they have limited applicability in cases where dynamic descriptions are required. They emphasize the flow of data through system objects rather than the interactions among these objects.

Languages using a message-passing model (Robson 1981a, Robson 1981b, McArthur 1984, Goldberg 1984a, Goldberg 1984b), unlike the ERA-based languages, are able to express system dynamics through a series of message communications. In this model, each object is defined by its attributes and the operations it can perform. Thus, for object A to interact with object B, object A SENDS object B a message requesting that object B perform an operation. Object B RECEIVES the message and performs the requested operation, which may include sending a message to another object or back to object A. This SEND and RECEIVE format of object interaction has potential for allowing description of some of the concepts needed in simulation models such as timing constraints, concurrency, synchronization, and environmental interaction.

The choice between an ERA-based or a message-based language is very dependent on the type of system to be specified. Both models encapsulate system behavior according to objects thereby easily localizing specification information. But the major advantage of using the object-oriented approach is that the resulting specification corresponds directly and naturally to the real system (Borgida 1985, p. 85).

3.1.3. Process Orientation

In a process-oriented specification language, a system is decomposed into processes, where a process may represent an object or an activity. Both static and dynamic descriptions of the system are possible. The static properties of each process are defined by enumerating its possible states, inputs, and outputs (Yeh 1984, Alford 1985). The dynamics of each process are described as a series of state transitions with each state transition producing certain responses and yielding a new process state. The resulting specification shows both data flow and control flow (Zave 1972) as well as the relationship of each process to its environment. Each process is assumed to operate in parallel with and asynchronous to the other processes (Zave 1984b).

The origins of many process-oriented specification languages can be traced back to the need to specify operating systems. Thus, these languages are well suited for applications involving complex controls and embedded systems (Zave 1982). Also the underlying stimulus-response model is actually a modified finite state machine thereby making it possible to use results from finite automata theory in analyzing the specification (Alford 1985).

The process-oriented languages listed in Table 2 have several weaknesses:

- They excessively employ formal notation that is difficult to understand and learn.
- They may be difficult to apply to systems other than the types mentioned in the previous paragraphs.
- The resulting specification may not be naturally hierarchical nor modular (Alford 1985, Zave 1982).
- The languages, with the exception of RSL, are still in the development stages.

3.1.4. The Review Summary

A review of the literature on software specification languages reveals a variety of approaches to specification and numerous formalisms. Examining the basic descriptive unit and the phase of the life-cycle to be supported enables an effective classification of software specification languages. Most of the languages reviewed are developed for certain application domains; thus, they are easier to use and produce a "better" specification when applied to problems in these domain areas (Zave 1984b).

3.2. Simulation Model Specification Languages

The need for model specification and model documentation has not escaped notice in the simulation research community (see (Nance 1977)). However, a consensus as to the precise nature of this need and to the proper means for meeting it is clearly lacking. The main challenge in the design of any Simulation Model Specification and Documentation Language (SMSDL) is how to express model dynamics (Nance 1977) because the dynamic dependencies in a simulation model are inherently complex (Nance 1984). Examples of SMSDLs are DELTA, GEST, ROSS, a SIMULA derived SMSDL proposed by Frankowski and Franta, and the Condition Specifications. Each is briefly described.

3.2.1. DELTA

The DELTA language (Holbaek-Hanssen 1977, Handlykken 1980) represents one of the most extensive SMSDL efforts. An object-oriented specification language, DELTA uses a mixture of formal and informal constructs to describe a system as a series of objects, with attributes, states, and the actions each object performs. The language encourages the production of a specification which is appropriate for communicating details of system behavior to an audience with diverse backgrounds. Its SIMULA-like constructs enable the expression of simulation model dynamics. Specifically, one can describe time, events, time-consuming actions, instantaneous actions, parallel actions, state changes, and activity interruptions.

Initially, the developers of DELTA had planned to design three languages:

(1) a general systems description language, DELTA

(2) a high level programming language, BETA

(3) and a systems programming language, GAMMA.

As of 1980, only DELTA exists, and no mention is made of GAMMA. The developers plan to use DELTA as the overall system development language and to translate the DELTA system specification into the executable programming language BETA.

3.2.2. GEST

Based on the principles of system theory, GEST (Ören 1979, Ören 1984) provides a conceptual framework for specifying discrete, continuous, and memoryless models. The static model structure consists of the definition of input, output, and state variables. The dynamic model structure is described via a series of state transition functions. A GEST specification also incorporates Zeigler's (Zeigler 1984) experimental frame concept. An experimental frame is the specification of the input and output data necessary for execution of the simulation model to answer the questions posed in the study objectives (Zeigler 1984, p. 22). Information specified in a GEST experimental frame includes such items as the basic unit of time, simulation termination conditions, state variable initializations, and data collection details (Ören 1984, p. 316). Future plans are for GEST to be integrated into a computer-assisted modeling system (Ören 1984).

3.2.3. ROSS

Developed by the RAND Corporation, ROSS (Klahr 1980, Faught 1980, McArthur 1981, McArthur 1984), represents one of the first attempts to combine artificial intelligence and simulation. It is designed for developing interactive knowledge-based event-driven simulators for particular applications. Currently, two combat-oriented simulators (SWIRL (Klahr 1982) and TWIRL (Klahr 1984)) have been developed. ROSS is not a specification language, but rather it is a LISP-based system which supports the development of the above mentioned simulators. An example of an object-oriented message-passing language, ROSS demonstrates how AI and simulation can be combined.

3.2.4. SMSDL — Frankowski and Franta

The proposal for a process-oriented simulation model specification and documentation language for specifying discrete-event simulations comes from Frankowski and Franta (Frankowski 1980). In this SMSDL, the model element (or object) is the primary unit of specification. Each element is described by attributes, axioms, and a scenario. Of these three, the scenario is of most interest because it contains the details of an element's behavior throughout the life of the model.

The scenario describes the changes of state in the model which trigger the behavior of each element. These changes of state may be invoked by the element itself or other model elements. The SMSDL does not differentiate between behavior dependent on a change in state or a change in time. Time is simply viewed as an attribute which may produce a state change. The behaviors that are specified in a scenario are referred to as actions. Constructs are available in the SMSDL for expression of interruptible and noninterruptible actions, actions that have duration, concurrent actions, and actions that occur in a predefined order.

Strongly reflecting the SIMULA influence, the SMSDL nevertheless represents an attempt to produce a readable, English-like specification which is suitable for communicating with a diverse audience. But more importantly, the SMSDL addresses some of the difficulties in expressing model dynamics and offers some possible solutions.

3.2.5. Condition Specification

The Condition Specification (CS) is suggested by Overstreet (Overstreet 1982, Overstreet 1985). The CS clearly achieves the stated goal of a world-view-independent specification. In addition to the property of world view independence, a CS has the following properties: (Overstreet 1982, p. 5)

(1) Achieves independence from implementation languages.

(2) Creates an analyzable specification.

(3) Assists in detecting errors in the specification.

(4) Permits translation to a specification in any of the three traditional world views.

(5) Enables any discrete-event simulation model to be completely described.

(6) Encourages successive refinement and elaboration of the specification.

(7) Produces a specification in which the model elements have direct correspondence to the elements of the modeled system.

A Condition Specification consists of three components (Overstreet 1985, p. 196). The *interface specification* identifies the input and output attributes of the model, and it is through these attributes that the model communicates with its environment. The *report specification* describes the data to be produced as a result of an execution of the simulation. The *specification of model dynamics* is the most important CS component, achieved through expanding the object specifications (static model structure) into a set of transition specifications. The transition specifications describe the changes that occur in the model as a series of Condition Action Pairs (CAPs), a rule composed of a boolean condition and an action to be performed whenever the condition is true. A boolean condition may be based upon time, state, or both. An action represents the model's response to the boolean condition and may include such responses as changing the value of an object attribute, scheduling another action to occur in the future, or terminating the simulation.

4. The Role of Simulation Model Specification

The simulation model life cycle of Figure 7 indicates the scope of items which may require specification in a large simulation project. While many are similar to those of any large software project, several are either unique to simulation projects or occur frequently in simulation but rarely in more general software projects.

4.1. Differences in Simulation Specification and Software Specification

Several aspects of a simulation project for which specifications may be useful but which are not usually part of a general software project are listed below; the list emphasizes the differences in the problems of general software projects and simulation projects.

- An explicit *reliance on modeling* is necessary. Conceptual, communicative, programmed, and experimental models must be created and tested. Preference must be given to simple models which still satisfy the study objectives, but often it is not clear how much detail is necessary. Abstraction requires insight.

- *Real data collection requirements* for establishment of model parameters, both static and dynamic, model input data, and output data for model validation.

- For many models, the data collected during a simulation run are observations in a statistical experiment. As such, the experimenter may be concerned with *experimental design* and making appropriate inferences from the data collected.

- *Validation procedures* can be significantly more difficult than for general software. If a model includes stochastic components, validation of output data may involve statistical testing. The problem is particularly difficult if no real system data are available for validating model output data.

- *Sensitivity analysis* may be required to determine how sensitive model output data are to simplifying assumptions and parameter estimates.

- Issues of *model credibility* must be addressed. While this is similar to assessment of software reliability in the general software domain, it is a persistent issue in the simulation domain.

- Often the *modeler*, a specialist in an application field, *is also the programmer*. While this occurs in general software development, it is significantly more prevalent in the simulation area.

- A *reliance on statistical analysis* is necessary for model definition, model validation, and proper interpretation of model results.

4.2. Similarities With General Software Requirements Specifications

Simulation projects have many problems which are identical to those of any large software projects. The hope is that many of these can be addressed with the use of more effective general software specification tools.

- The *management problems* for large models are similar to those of any large software project. The *lack of good costing tools* is particularly acute here. Reliable costing and scheduling will not be achieved without better product specifications.

- A model should only be as detailed as necessary to produce needed reliable results. Since, for some models and study objectives, the degree of required detail cannot be known when the project is started, some projects are hard to plan, schedule, and budget since *accurate, complete specifications cannot be formulated without experimenting* with some version of the completed product. This occurs in some general software projects.

- *Technology transfer* is slow in both areas. Costs of reeducation limits utilization even when significant improvements are possible. Utilization of new technologies in large projects is appropriately perceived as risky since new approaches are not proven effective.

- *Reusability* is not well supported in either area.

- The *complexity problems will not be eliminated* with the creation of more effective technologies. As more effective means are found to deal with existing complex systems, construction of new systems with additional complexity will be perceived as cost effective.

- Current technologies induce significant overheads which are *not cost effective for small, limited-use projects.* If is often difficult to anticipate project size or length of use.

For large software projects, individuals with different responsibilities will make different uses of the requirements specification (Berg 1982, p. 3). Some typical roles and responsibilities are given in Table 3.

Table 3. Roles and Responsibilities of Users of Specifications

Role	**Responsibility**
Manager	Prepare (and justify) cost estimates, time schedules, and work assignments.
Validator	Confirm that the specification addresses the "real problem."
Verifier	Confirm that the implementation satisfies the specification.
Implementor	Produce an efficient, well-designed implementation.

Since the communicative model serves many of the same functions as a requirements specification, similar uses are made of it. Thus, it is a crucial component of the specification process.

4.3. SPLs as Specification Languages

One objective of the preceding discussions is to indicate the variety of uses which may be made of a communicative model specification. SPLs, because they are designed for the creation of programmed models, are appropriate for specifying only limited number of the items requiring specification in a large simulation project. Considering the roles listed in Table 3, the communicative model specification is of key importance to the individuals in the roles of manager, validator, verifier, and implementor. Use of an SPL for specification of a communicative model may well assist the implementor and the verifier; however, its effectiveness in meeting the manager's needs for planning and costing is questionable. Also questionable is its ability to assist the validator in confirming that the specification addresses the "real problem." A central thrust in the software engineering community is to find effective support for early stages of the software life cycle. By supporting later rather than early stages, use of SPLs as specification languages runs counter to these efforts.

In addition many widely used SPLs provide too few levels of abstraction for large models. For example, if a large model is described in GPSS, the language provides very limited features for allowing a reader to understand the model first in broad terms, and then to proceed to understand the model in progressively more detailed layers. SPLs which work effectively for describing small systems usually do not scale up to handle large communicative models effectively. This is an unfortunate feature of the iconic simulation modeling tools.

The world view of a particular SPL may be a hindrance rather than assistance in specifying a particular model. (Henriksen 1981)

Use of an SPL for specification of communicative models encourages premature concern with implementation techniques. Creation of a programmed model requires addressing three distinct aspects of the resulting executable program:

(1) The behavior of the model must be specified.

(2) Techniques for data collection during execution must be determined and perhaps the analyses to be performed on model termination.

(3) An efficient implementation must be created.

Effective solutions are confounded when problems of model behavior, data collection and analysis, and efficient implementation are addressed simultaneously. Most SPL provides mechanisms for dealing with all three, but they are distinct, even if related, problems.

5. Conclusions

The summaries of sections 2 and 3 on software life cycles and specification languages give several observations about the state of software engineering.

(1) A consensus exists in the importance of specification in a large software project.

(2) A lack of consensus exists on how one should specify the requirements of a large software product.

(3) A lack of consensus exists on the proper steps to be used, that is, the methodologies, in the

creation of a large software product.

A variety of specification languages and methodologies have been proposed and several are commercially available. A serious short-coming has been the lack of effective evaluation of competing specification languages, methodologies, and now workstation-based environments which claim to address these problems. Most "evaluations" are by product developers whose objectivity is suspect. Celko at al. (Celko 1983) describe one large independent, and inconclusive, evaluation which specified the same system with three different languages. Stoegerer provides an informal summary, by way of a matrix, of the features of a variety of specification languages (Stoegerer 1984). Recent suggestions on how these evaluations might be performed have been made by (Arthur 1986) and (Welderman 1987) but it is clear that the proposed evaluation methodologies must evolve with experience.

Thus the simulation community cannot yet look to the software engineering community for a proven reliable solution; specification languages and development methodologies are areas of much active research. In addition, how effectively to utilize the powerful networked work stations now available at modest costs have added to this uncertainty.

Several other related problems in the simulation area are common to general software specification. Effective solutions to these problems have not been found for general software.

- Management needs for costing, budgeting, scheduling, and tracking large projects are poorly addressed.
- Size and anticipated use effect the overhead which should be acceptable, but are these are difficult to anticipate.

We more often err by underestimating the long-term benefits of creating items such as a communicative specification, but as stated in *The Soul of a New Machine,* "Not everything worth doing is worth doing well." We do a poor job of distinguishing.

Several items which should be specified in a large simulation project are either unique to simulation or exist here to a significantly greater degree then for general software systems. These are summarized in Section 4. Tools and methodologies to assist in these areas will not come from the general software community.

Experience from software engineering contributes more in suggesting the directions which should be taken in the simulation area than in providing solutions. We recognize the important and multiple role of specifications. We recognize the advantage in identifying the steps which should occur in the simulation model life cycle and the cyclical nature of the process. We recognize the importance of providing more effective tools, methodologies and environments to assist in this process.

REFERENCES

Alford, M. W. (1977). "A Requirement Engineering Methodology for Real-time Processing Requirements," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, January, pp. 60-69.

Alford, M. W. (1985). "SREM at the Age of Eight: The Distributed Computing Design System," *IEEE Computer,* Vol. 18, No. 4, April, pp. 36-46.

Arthur, J. D., Nance, R. E., and Henry, S. M. (1986). "A Procedural Approach to Evaluating Software Development Methodologies: The Foundation," Technical Report SRC-86-008, Systems Research Center, Virginia Tech, Blacksburg, VA 24061, September. 26 pp.

Balci, O. (1986). "Guidelines for Successful Simulation Studies," Technical Report TR-85-2, Department of Computer Science, Virginia Tech, Blacksburg, VA., Sept.

Balzer, R. (1980). "An Implementation Methodology for Semantic Database Models," In: *Entity-Relationship Approach to Systems Analysis and Design,* P. P. Chen, (ed.), North-Holland Publishing Company, Amsterdam, pp. 433-444.

Balzer, R. M., Goldman, N. M., and Wile, D. S. (1982). "Operational Specification as the Basis for Rapid Prototyping," *ACM SIGSOFT Software Engineering Notes,* Vol. 7, No. 5, December, pp. 3-16.

Balzer, R., Cheatham, T. E., Jr., and Green, C. (1983). "Software Technology in the 1990's: Using a New Paradigm," *IEEE Computer,* Vol. 16, No. 11, November, pp. 39-45.

Bell, T. E., Bixler, D. C. and Dyer, M. E. (1977). "An Extendable Approach to Computer-aided Software Requirements Engineering," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, January, pp. 49-60.

Benington, H. D. (1956). "Production of Large Computer Programs," In: *Proceedings of ONR Symposium on Advanced Programming Methods for Digital Computers* (Washington, D.C., June 28-29). Office of Naval Research Symposium Report ACR-15. Also available in *Annals of the History of Computing 5,* 4 (Oct. 1983), 350-361.

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

Berzins, V. (1985). "Analysis and Design in MSG.84: Formalizing Functional Specifications," *IEEE*

Transactions on Software Engineering, Vol. SE-11, No. 8, August, pp. 657-670.

Boehm, B. W. (1976). "Software Engineering," *IEEE Transactions on Computers C-25*, 12 (Dec.), 1226-1241.

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

Borgida, A., Greenspan, S., and Mylopoulos, J. (1985). "Knowledge Representation as the Basis for Requirements Specifications," *IEEE Computer*, Vol. 18, No. 4, April, pp. 82-91.

Celko, J., Davis, J. S., and Mitchell, J. (1983). "A Demonstration of Three Requirements Language Systems," *ACM SIGPLAN Notices*, Vol. 18, No. 1, pp. 9-14, Jan.

Chen, P. P. (1976). "The Entity-relationship Model — Toward a Unified View of Data". *ACM Transactions on Database Systems 1*, 1 (March) pp. 9-36.

Davis, A. M. (1982). "The Design of a Family of Application-Oriented Requirements Languages," *IEEE Computer*, Vol. 15, No. 5, May, pp. 21-28.

Davis, C. G. and Vick, C. R. (1977). "The Software Development System," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January, pp. 69-84.

Distase, J. R. (1980). "Software Management - A Survey of the Practice in 1980," In: *Proceedings of the IEEE 68*, 9 (Sept.), 1103-1119.

Faught, W. S., Klahr, P., and Martins, G. R. (1980). "An Artificial Intelligence Approach to Large-Scale Simulation," *Summer Simulation Conference*, Seattle, Washington, (August 25-27), pp. 231-235.

Feather, M. S. (1983). "Reuse in the Context of a Transformation Based Methodology," In: *Proceedings of the Workshop on Reusability in Programming*, Newport, R.I., (September 7-9), pp. 50-58.

Frankowski, E. N. and Franta W. R. (1980). "A Process Oriented Simulation Model Specification and Documentation Language," *Software — Practice and Experiences*, Vol. 10, No. 9, September, pp. 721-742.

Freeman, P. (1983). "Fundamentals of Design," In: *Tutorial on Software Design*, P. Freeman and A. Wassermann, (eds.), IEEE Computer Society Press, pp. 2-22.

Gehani, N. (1982). "Specifications: Formal and Informal — A Case Study," *Software - Practice and Experience*, Vol. 12, pp. 433-444.

Giddings, R. V. (1984). "Accommodating Uncertainty in Software Design," *Communications of the ACM 27*, 5 (May), 428-343.

Goldberg, A., and D. Robson (1984a) *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1984.

Goldberg, A. (1984b) *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.

Goldman, N. and Wile, D. (1980). "A Relational Database Foundation for Process Specification," In: *Entity-Relationship Approach to Systems Analysis and Design*, P. P. Chen, (ed.), North-Holland Publishing Company, Amsterdam, pp. 413-432.

Hamilton, M. and Zeldin, S. (1976) "Higher Order Software - A Methodology for Defining Software," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, January, pp. 9-32.

Hamilton, M. and Zeldin, S. (1983). "The Relationship Between Design and Verification," In: *Tutorial on Software Design Techniques*, P. Freeman and A. Wassermann, (eds.), IEEE Computer Society Press, pp. 641-668.

Handlykken, P. and Nygaard, K. (1980). "The DELTA System Description Language Motivation, Main Concepts, and Experience from Use," In: *Software Engineering Environments*, H. HUNKE, (ed.), North-Holland Publishing Company, Amsterdam, pp. 173-190.

Heacox, H. C. (1979). "RDL - A Language for Software Development," *ACM SIGPLAN Notices*, Vol. 14, December, pp. 71-79.

Henriksen, J. O., (1981). "GPSS — Finding the Appropriate World View," In: *Proceedings of the Winter Simulation Conference*, Atlanta, GA, (9-11 December), pp. 505-516.

Holbaek-Hanssen, E., Handlykken, P. and Nygaard, K. (1977). *System Description and the Delta Language*, Report No. 4, Norwegian Computing Center, Oslo.

Klahr, P., and Faught, W. S. (1980). "Knowledge-Based Simulation," In: *Proceedings of the First Annual Conference of the American Association for Artificial Intelligence*, Stanford, CA, pp.181-183.

Klahr, P., McArthur, D., Narain, S., and Best, E. (1982). *SWIRL: Simulating Warfare in the ROSS Language*, Rand Report N-1885-AF, The Rand Corporation, Santa Monica, CA, September.

Klahr, P., et al. (1984). *TWIRL: Tactical Warfare in the ROSS Language* Rand Report R-3158-AF, The Rand Corporation, Santa Monica, CA, October.

Lehman, M. M. (1980). "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE 68*, 9 (Sept.), 1060-1076.

Lehman, M. M. (1984a). "A Further Model of Coherent Programming Processes," In In: *Proceedings of Software Process Workshop* (Egham, Surrey, U.K., Feb. 6-8). IEEE CS, Los Angeles, CA., IEEE Catalog Number 84CH2044-6, pp. 27-35.

Lehman, M. M., Stenning, V., and Turski, W. M. (1984b). "Another Look at Software Design Methodology," *ACM Software Engineering Notes 9*, 2 (Apr.), 38-53.

Liskov, B. H. and Zilles, S. N. (1975). "Specification Techniques for Data Abstraction," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March, pp. 7-19.

Liskov, B. (1984), "Programming Language Issues for the 1980s," Panel Discussion Comments from *SIGPLAN Notices*, **19**, 8 (August), 51-61.

London, P. E. and Feather, M. S. (1982). "Implementing Specification Freedoms," In: *Science of Computer Programming*, Vol. 2, North-Holland Publishing Company, Amsterdam, pp. 91-131.

Martin, J. and McClure, C. (1985a). *Diagramming Techniques for Analysts and Programmers*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Martin, J. (1985b). *System Design From Provably Correct Constructs*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

McArthur, D. and Sowizral, H. (1981). "An Object-Oriented Language for Constructing Simulations," In: *Proceedings of the International Joint Conference on Artificial Intelligence*, Vancouver, Canada, pp. 809-814.

McArthur, D., Klahr, P., and Narain, S. (1984). *ROSS: An Object-Oriented Language for Constructing Simulations*, Rand Report R-3160-AF, The Rand Corporation, Santa Monica, CA, December.

Nance, R. E. (1977). "The Feasibility of and Methodology for Developing Federal Documentation Standards for Simulation Models," Final Report to the National Bureau of Standards, Department of Computer Science, Virginia Tech, Blacksburg, VA, June.

Nance, R. E. (1981). "Model Representation in Discrete Event Simulation: The Conical Methodology," Technical Report CS81003-R, Department of Computer Science, Virginia Tech, Blacksburg, VA, March 15.

Nance, R. E. (1984). "Model Development Revisited," In: *Proceedings of the 1984 Winter Simulation Conference*, Dallas, Texas, (November 28-30), pp. 75-80.

O'Brien, P. D. (1983). "An Integrated Interactive Design Environment for TAXIS," *SOFTFAIR*, Arlington, VA, (July 25-28), pp. 298-306.

Ören , T. I. and Zeigler, B. P. (1979). "Concepts for Advanced Simulation Methodologies," *Simulation*, Vol. 32, No. 3, March, pp. 69-82.

Ören , T. I. (1984). "GEST - A Modelling and Simulation Language Based on System Theoretic Concepts," In: *Simulation and Model-Based Methodologies: An Integrative View*, T.I. Ören , B.P. Zeigler, M.S. Elzas, (eds.), Springer-Verlag, NY, pp. 281-335.

Overstreet, C. M. (1982). *Model Specification and Analysis for Discrete Event Simulation*, PhD Dissertation, Department of Computer Science, Virginia Tech, Blacksburg, VA, December.

Overstreet, C. M. and Nance, R. E. (1985). "A Specification Language to Assist in Analysis of Discrete Event Simulation Models," *Communications of the ACM*, Vol. 28, No. 2, February, pp. 190-201.

Pohoski, M. W. (1981). "A Top Level Description of the Ship Combat System Simulation". Naval Ocean Systems Center Report (jointly with the Naval Weapons Center and the Naval Surface Weapons Center).

Riddle, W. E., et al. (1978a). "A Descriptive Scheme to Aid the Design of Collections of Concurrent Processes," In: *Proceedings AFIPS National Computer Conference*, Anaheim, CA, June, pp. 549-554.

Riddle, W. E., et al. (1978b). "Behavior Modeling During Software Design," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 4, July, pp. 283-292.

Riddle, W. E. (1979). "An Event-Based Design Methodology Supported by DREAM," In: *Tutorial on Software Design Techniques*, P. Freeman and A. Wassermann, (eds.), IEEE Computer Society Press, (1983), pp. 378-392.

Riddle, W. E. (1980). "An Assessment of DREAM," In: *Software Engineering Environments*, Horst HUNKE, ed., North-Holland Publishing Company, Amsterdam, pp. 191-221.

Robson, D. (1981a). "Object-Oriented Software Systems," *BYTE*, Vol. 6, No. 8, August, pp. 74-86.

Robson, D. and Goldberg, A. (1981b). "The Smalltalk-80 System," *BYTE*, Vol. 6, No. 8, August, pp. 36-48.

Royce, W. W. (1970). "Managing the Development of Large Software Systems: Concepts and Techniques," In: *Proceedings of the 1970 WESCON* (Los Angeles, CA., Aug. 25-28), Western Periodicals Co., North Hollywood, CA., pp. A/1/1-9.

Scheffer, P. A., Stone, A. H. III, and Rzepka, W. E. (1985). "A Case Study of SREM," *IEEE Computer*, Vol. 18, No. 4, April, pp. 47-54.

Silverberg, B. A. (1981). "An Overview of the SRI Hierarchical Development Methodology," *Software Engineering Environments*, Horst HUNKE, ed., North-Holland Publishing Company, Amsterdam, pp. 235-252.

Stoegerer, J. K. (1984). "A Comprehensive Approach to Specification Languages," *Australian Com-*

puter Journal, Vol. 16, No. 1, February, pp. 1-13.

Teichroew, D. and Hershey, E. A. III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 41-48.

Teichroew, D., et al. (1980). "Application of the Entity-Relationship Approach to Information Processing Systems Modeling," In: *Entity-Relationship Approach to Systems Analysis and Design*, P. P. Chen, (ed.), North-Holland Publishing Company, Amsterdam, pp. 15-38.

Wassermann, A. I. (1983). "Information System Design Methodology," In: *Tutorial on Software Design*, P. Freeman and A. Wassermann, (eds.), IEEE Computer Society Press, pp. 43-62.

Welderman, N. H., Habermann, A. N., Borger, M. W., and Klein, M. H., (1987). "A Methodology for Evaluating Environments," *ACM SIGPLAN Notices*, Vol. 22, No. 1, pp. 199-207, Jan. (*Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*)

Winters, E. W. (1979). "An Analysis of the Capabilities of PSL: A Language for System Requirements and Specifications," *IEEE COMPSAC*, Chicago, IL, November, pp. 283-288.

Yeh, R. T., Zave, P., Conn, A. P., and Cole, G. E., Jr. (1984). "Software Requirements: New Directions and Perspectives," In: *Handbook of Software Engineering*, C.R. Vick and C.V. Ramamoorthy, (eds.), Van Nostrand Reinhold Company, New York, pp. 519-543.

Zave, P. (1979). "A Comprehensive Approach to Requirements Problems," *IEEE COMPSAC*, Chicago, IL, November, pp. 117-122.

Zave, P. (1982). "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 3, May, pp. 250-269.

Zave, P. (1984a). "The Operational Versus the Conventional Approach to Software Development," *Communications of the ACM*, Vol. 27, No. 2, February, pp. 104-118.

Zave, P. (1984b). "An Overview of the PAISLey Project - 1984," *ACM SIGSOFT Software Engineering Notes*, Vol. 9, No. 4, July 1984, pp. 12-19.

Zeigler, B. P. (1984). "Theory and Application of Modeling and Simulation: A Software Engineering Perspective," In: *Handbook of Software Engineering*, C.R. Vick and C.V. Ramamoorthy, (eds.), Van Nostrand Reinhold Company, NY, pp. 1-25.

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SRC 87-001 / (CS TR-87-7) — 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Specification Languages: Understanding Their Role in Simulation Model Development	5. TYPE OF REPORT & PERIOD COVERED Interim
	6. PERFORMING ORG. REPORT NUMBER SRC 87-001 / (CS TR-87-7)
7. AUTHOR(s) C. Michael Overstreet, Richard E. Nance, Osman Balci, and Lynne F. Barger	8. CONTRACT OR GRANT NUMBER(s) N60921-83-G-A165
9. PERFORMING ORGANIZATION NAME AND ADDRESS Systems Research Center and Department of Computer Science, Virginia Tech, Blacksburg, VA 24061	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Sea Systems Command, SEA61E, Washington, D.C. 20362	12. REPORT DATE December 1986
	13. NUMBER OF PAGES 31
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Naval Surface Weapons Center, Dahlgren, VA 22448	15. SECURITY CLASS. *(of this report)* Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*
This report is distributed to scientists and engineers at the naval Surface Weapons Center and is made available on request to other Navy scientists. A limited number of copies is distributed for peer review.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*
To Navy research and development centers and university-based laboratories.

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*
Software Engineering: Requirements/Specifications--Languages; Simulation and Modeling; Model Life Cycle, Simulation Model Specification, Software Life-cycle Models

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*
Current software specification techniques and specification languages are reviewed, emphasizing research activities in software specification languages. Alternate software life cycle models are described and compared to a simulation life cycle model. The importance of constructing a model specification before creating a programmed model is emphasized. Disadvantages in using simulation programming languages as model specification languages are discussed. The multiple uses which are made of a model specification are presented; these uses correspond to the alternate uses made of a requirements

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

specification for general software. To evaluate where specification tools for general software will be effective for simulation modeling, both areas where the simulation life cycle corresponds to a general software life cycle and areas in which they differ are characterized. Important conclusions are that: (1) specification methodologies are highly dependent on the phase of the software life cycle and the use to be made of the specification, (2) both general software and simulation communities lack a clear perception of the proper form for specifications, and (3) evaluation of proposed methodologies and tools is adequate.

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE(*When Data Entered*)