Technical Report SRC-94-002[1]

# Simulation Model Specification: On the Role of Representation in the Model-Centered Sciences

*Ernest H. Page*
*Richard E. Nance*

Systems Research Center
and
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106

October 17, 1995

[1]Cross Referenced as Technical Report TR-94-04, Department of Computer Science, Virginia Tech.

## Abstract

The old adage that "a problem correctly formulated is half solved" is rarely challenged. Model-centered problem solving relies critically on a correct model. Complete, clearly stated assumptions, a precise statement of objectives, and an adequate representation of the model of the system under study are readily accepted as responsibilities of the modeler. But just how important are both the process and product of model representation? The modeler's responsibilities in producing a *correct* model can be either significantly aided or drastically inhibited by the representational mechanism – the language – being used. The comprehensibility of the model, subsequent extension, adaptation, or maintenance of the model, are highly dependent on the representational language. Language for model representation is therefore an indispensable tool. The pivotal role of this tool is described through its relationship to modeling methodology, model development environments, and automated model diagnosis. The need for an emphasis on the capabilities for conceptual expressiveness, in contrast with execution efficiency is discussed. While software engineering research clearly advocates this redefinition of emphasis, we observe that much of the research in the rising disciplines within parallel computation appears to ignore the lessons of this history.



**Categories and Subject Descriptors:** I.6.5 [**Simulation and Modeling**]: Model Development – *Modeling methodologies*; I.6.7 [**Simulation and Modeling**]: Simulation Support Systems – *Environments*

**General Terms:** Model specification, Conical Methodology, Condition Specification, Simulation Model Development Environment.

# 1 Introduction

The old adage that "a problem correctly formulated is half solved [26]" is rarely challenged. Model-centered problem solving relies critically on the correctness of the model. As such, a complete, clearly stated set of assumptions, a precise statement of objectives, and an adequate representation of the model of the physical system under study are readily accepted as responsibilities of the modeler.

Less well recognized is the necessity for the model to be understood. Perhaps history is most to blame for this fact; model comprehension is almost trivial in a problem-solving domain consisting of a single analyst, reporting to a single decision maker, whose basic understanding of linear programming is sufficient to appreciate the capabilities and limitations of the solution technique. While this scenario is exaggerated in its simplicity, the contrast between model derived decisions in the 1960's versus today is stark. A modeling group may be dispersed within a company, across different companies, located on different continents, and yet be realistically expected to produce a model whose structure can be verified by yet another group and whose results can be validated by still another group. And, ultimately, the model and its results may be utilized by an almost equally diffused set of decision makers.

Why should professionals in operations research, computer sciences, management sciences, decision sciences, etc. be concerned with model representation? The likelihood that this question – almost a direct quote from a referee's report of some eight years ago – might still be asked today, even from members of this audience, is the motivation for this paper.

The rhetorical answer is that the modeler's responsibilities in producing a *correct* model can be either significantly aided or drastically inhibited by the representational mechanism(s) being used: the language(s). The comprehensibility of the model, so important in verification and validation as well as the credibility assessment rendered by those intending to use the model for decision support, is highly dependent on the representational language. Subsequent extension, adaptation, or maintenance (all functions that are better grouped under the term *model evolution*) are also highly dependent on the representational language. Language for model representation is therefore an indispensable tool; what group purporting to be professional would abdicate the responsibility for defining, designing or creating its tools?

Using the evolution of discrete event simulation as an example, this paper describes a role for model representation in a model-centered problem solving domain. We consider the influence of model representation within a life cycle model of a simulation study; an influence whose pervasiveness often goes unrecognized. A methodology and environment supporting the life cycle – with deference to the key role of model representation within it – are briefly presented. Also described is a simulation model specification and documentation language, the condition specification (CS), which satisfies many, although not all, of the requirements for *next generation* model representations.

## 2 Shifting From a Program View

According to Nance [16], the past decade has seen a shift in the focus of the discrete event simulation community from a *program*-centric view of the simulation process to a *model*-centric view.

Motivating this shift has been an evolving recognition that programming language representations of simulation models necessarily contain many implementation-related details that obfuscate the clear enunciation of model behavior. Furthermore, and potentially the most damaging, the use of a particular language has direct, often hidden, influences on the structure of the model formulation itself. To illustrate this point, consider the classical machine interfence problem in which a technician monitors and repairs a set of machines that fail intermittently. A SIMULA implementation of this model may contain a description of the *lifetime* of a machine class object and a technician class object – a machine operates for some time, fails, waits for repair, and then repeats the cycle; a technician detects a machine failure, travels to the machine and repairs it. An efficient GPSS implementation of this model is very different: defining the technician as a static facility and defining machine failures as transactions which queue for the technician facility. And a SIMSCRIPT implementation of this same model can adopt an entirely different view by describing the model behavior that corresponds to identifiable *events* in the model, such as a machine failure or an end of repair.

This example raises the question: which implementation provides the "most natural" description of the machine interference problem? Obviously no definitive answer to this question exists; different people often view the same thing in different ways. Clearly though, a representation (GPSS) that characterizes machine failures as "moving" objects does not conform with the physical reality. Admittedly, a limited model – in numbers of objects and the interactions among them – can be comprehensible in any programming language; however the comprehensibility drops drastically with increases in model size and complexity. For instance, event descriptions (SIMSCRIPT) that involve multiple objects can become exceedingly complex.

The problem illustrated by this example is that of the *conceptual framework* within discrete event simulation modeling.

## 2.1 The programming language impedance

While no etiology of software errors exists, a consensus among many researchers is that most arise as the result of a poor mesh between the models of problems as they form in the mind (or minds) of a modeler (modeling team), and the models provided by extant programming languages and techniques.

The developers of simulation programming languages (SPLs) sought to close this conceptual distance through the provision of a *conceptual framework* (or "world view") within the language. The conceptual framework provides a modeler with a means to construct a *mental picture* of the model. According to the theory, if the model in the modeler's mind and the SPL utilize the same conceptual framework, the distance is closed.

The traditional conceptual frameworks can best be described as providing a view of a system through varying *localities* [22, p. 164]. For instance, the behavior of a system can be modeled according to the times at which things "happen" (the *event scheduling* world view; *locality of time*), according to the state precondition on the occurrence of something happening (the *activity scanning* world view; *locality of state*), or as an ordered sequence of actions performed on or by a given model object (the *process interaction* world view; *locality of object*). In his thesis, Derrick [9] classifies thirteen conceptual frameworks and identifies both positive and negative aspects of their influence on model representation.

The provision of conceptual frameworks within simulation programming languages ostensibly affords significant benefits for modeling as compared to general purpose languages. These conceptual bridges are not without their drawbacks however – the tendency to use the language best known by the modeler often results in a contrived "fitting" of the natural model description into the form provided by the simulation programming language, serving only to recreate the original impedance problem once removed.

## 2.2 The program as a communicative model

As identified in [17] a simulation programming language must provide:

1. generation of random numbers, so as to represent the uncertainty associated with an inherently stochastic model,
2. process transformers, to permit uniform random variates obtained through the generation of random numbers to be transformed to a variety of statistical distributions,
3. list processing capability, so that objects can be manipulated or created and deleted as sets or as members, added to and removed from sets,
4. statistical analysis routines, to provide the descriptive summary of model behavior so as to permit comparison with system behavior for validation purposes and the experimental analysis for both understanding and improving system operation,
5. report generation, to furnish an effective presentation of potentially large reams of data to assist in the decision making that initially stimulates the use of simulation, and
6. a time flow mechanism, to provide an explicit representation of time.

All the syntax concomitant with providing each of these capabilities, despite the best efforts of any SPL designer, clutters a programmed model with details that contribute nothing to the description of the behavior of the underlying system. Since ideally, in order to facilitate model analysis, we would like a description free of these and other *implementation* details, the need for higher level model representational forms becomes evident.

# 3 A Life Cycle Model for Simulation

A life cycle model of a simulation study is illustrated in Figure 1. The life cycle model contains ten phases (designated by ovals), ten processes (designated by dashed vectors) and eleven credibility assessment stages (represented by solid vectors).

The origins of the life cycle model pictured in Figure 1 can be traced to a group of guidelines identified by Nance as the "model life cycle [14]." This early form of the life cycle describes primarily those phases which comprise model development (the phases which form the circle in Figure 1). Nance and Balci [18] extend the original concepts into the form presented here.

The life cycle model assigns a critical role to model representation: adequate representation permits early detection of errors in the model (thus reducing the overall cost of error correction) and provides a potential target for model maintenance and enhancement. However, the life cycle model provides only a guiding structure for model development and use. The framework must be supported by one or more methodologies and realized within a modeling environment.

## 3.1 Methodology

A call for methodology in discrete event simulation had its earliest support from Lackner [12], Kiviat [11], Zeigler [28], and Nance [13, 14, 15] who proposes the conical methodology (CM). The CM reflects influences of the object-oriented (see [27]), entity-relation-attribute (see [8]) and automation-based (see [6]) paradigms, and has been demonstrated to support the larger principles of software engineering [1]. The CM structures the modeling process by defining two stages of model development: a top down model decomposition – the *model definition phase* – and a bottom-up model synthesis – the *model specification phase*. The phases in the methodology provide a pedagogical separation of the modeling process, but are neither intended to be executed independently or without iteration. The methodology encourages model development as a repet-

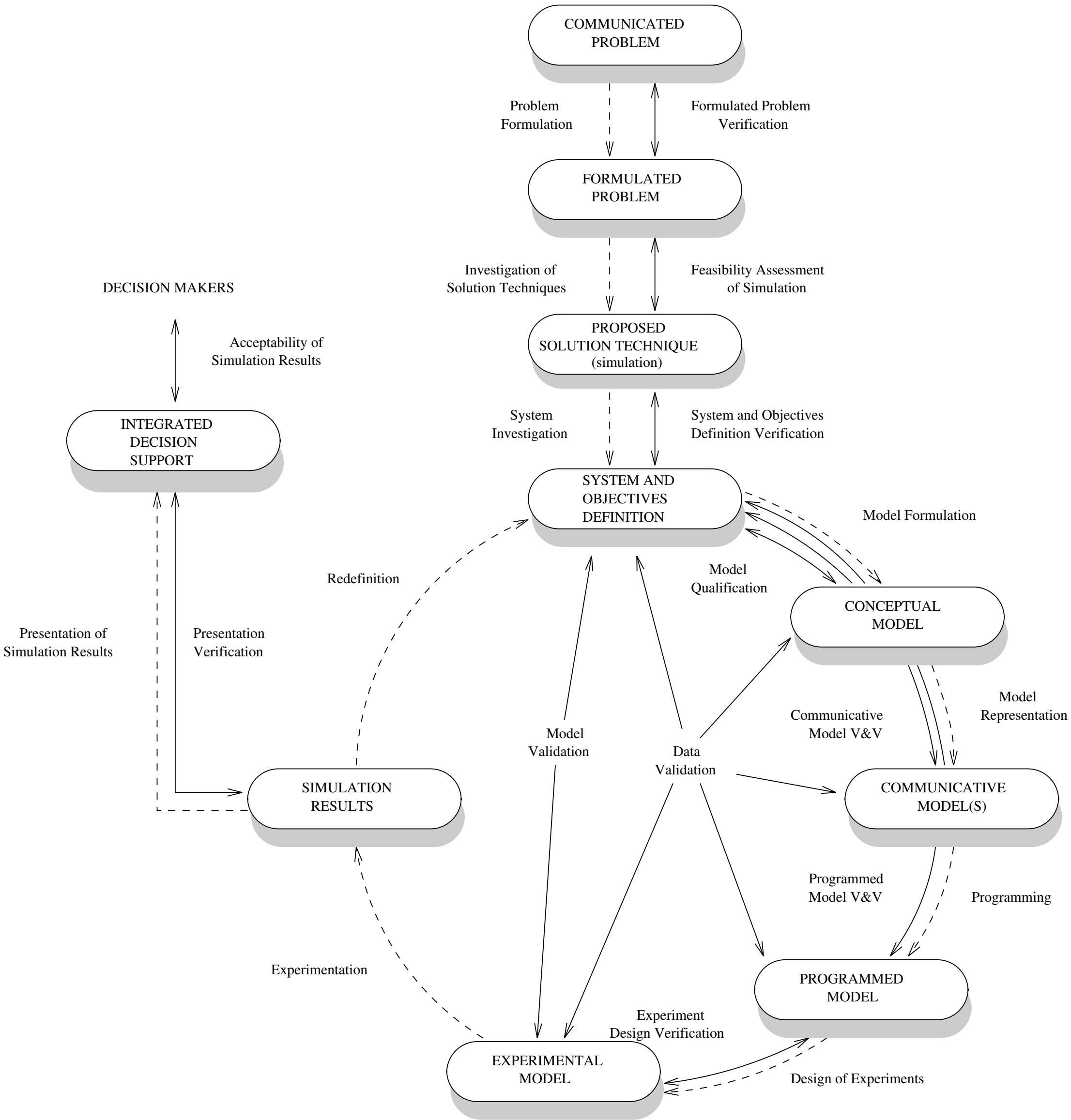


Figure 1: A Life Cycle Model of a Simulation Study

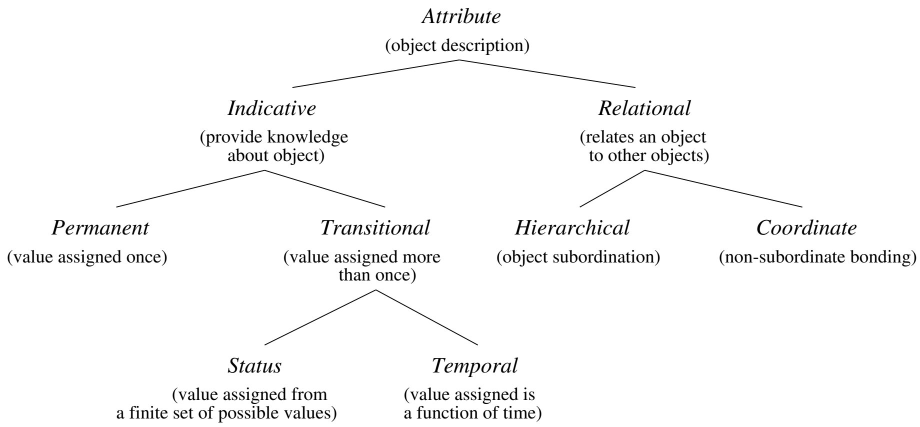


Figure 2: Conical Methodology Types.

itive cycling between definition and specification in a manner similar to successive refinement; no restrictions are placed on a modeler other than the requirement that object definition precede object specification for a given object.

### 3.1.1 Model definition phase

During the model definition phase, a modeler decomposes a model into objects and subobjects then names and types the attributes of these objects. This decomposition can be visualized as a tree with the root of the tree being the model itself (also an object) and the leaves representing the subobjects at the most detailed level. Thus, a modeler may view a model through any perspective of object-level granularity by focusing on a particular level of the development tree. In essence, during the model definition phase, the modeler describes the static aspects of the model.

The conical methodology advocates explicit typing. Explicit typing permits analysis of specifications to a greater extent than realizable in the absence of typing information [22]. The methodology prescribes that attributes, not necessarily objects, be typed. This attribute typing follows the taxonomy illustrated in Figure 2. To briefly summarize, *indicative* attributes describe an aspect of an object and may be classified as *permanent* or *transitional*. Permanent indicative attributes are assigned a value only once during model execution. Transitional indicative attributes receive multiple value assignments. Transitional indicative attributes come in two categories: *status* and *temporal.* Temporal transitional indicative attributes are assigned values that represent system time, whereas status transitional indicative attributes take on values that do not reflect system time. *Relational* attributes relate an object to one or more objects and may be classified as *hierarchical* or *coordinate.* Hierarchical relational attributes establish the subordination of one object to another; coordinate relational attributes establish a bond or commonality between two objects. In addition to typing, the dimensionality and range of an attribute may also be provided during model definition. The methodology does not preclude an attribute from having multiple types – most attributes that relate one object to another can often be described as providing indicative information about an object as well.

At least one attribute is associated with every model: the model indexing attribute. Typically this attribute is *system time* and is an attribute of the top level model object. As an example of CM typing consider the classical M/M/1 queueing model. The object `server` might have the attribute *status* – a status transitional indicative attribute which may assume the values *busy* or *idle*.

### 3.1.2 Model specification phase

The conical methodology advocates a bottom-up specification phase in which a modeler starts at the leaves of the model decomposition tree and describes the effects attributes of one object, by their value changes, have on attributes of other model objects – thereby describing model behavior. During the model specification phase, a modeler identifies the dynamic aspects of a model.

The CM enunciates the need for a simulation model specification and documentation language (SMSDL) but does not prescribe its form. The CM stipulates that the SMSDL should:

1. exhibit independence of the specification from implementational constraints of simulation programming languages,
2. allow expression of static and dynamic model properties,

3. facilitate model validation and verification, and

4. produce documentation as a by-product.

A specification language fulfilling these requirements is discussed in Section 4.

Under the CM, a partial specification of the attribute *status* of the object `server` in the M/M/1 example may take the form: "*status* of `server` equals *idle* when *end_of_repair* is true and *number_in_queue* is zero." This partial specification details the state change of the object `server` by indicating how one of its status transitional indicative attributes, *status*, changes value from *busy* to *idle*. Barger [2] asserts that status transitional indicative attributes are key to model specification under the conical methodology; Page [24] provides a reaffirmation of this claim.

### 3.2 Environment

The life cycle provides a guiding philosophy for model development and support. Methodologies provide a description of *how* to operate within the context of the life cycle model. The explicit structure for carrying out the precepts of a life cycle model and the methodologies supporting it is the domain of the model *environment*. Several environments for simulation model development and support have been proposed. We briefly describe one such effort here.

The Simulation Model Development Environment (SMDE) is the realization of the life cycle model illustrated in Figure 1, as supported by the conical methodology. The SMDE seeks to provide an integrated set of software utilities that offer automated support in the development, analysis, translation, verification, archival storage/retrieval, and management of discrete event simulation models, and thereby achieve the automation-based paradigm for simulation modeling through the evolutionary development of prototypes. Figure 3 depicts the architecture of the SMDE in four layers: (0) Hardware and Operating Systems, (1) Kernel SMDE, (2) Minimal SMDE, and (3) SMDEs.

Layer 0 is comprised of the prototype hardware, currently a SUN workstation (efforts are ongoing to move the SMDE to a NeXT platform). The kernel, Layer 1, integrates all SMDE tools

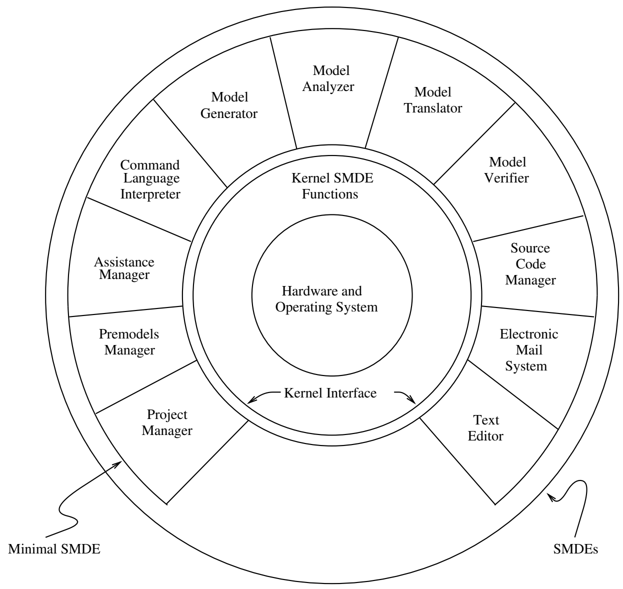


Figure 3: The SMDE Architecture.

into the software environment. Layer 2 provides a *comprehensive set* of tools which are *minimal* for the development and execution of a model. "Comprehensive" implies that the toolset is supportive of all model development phases; "minimal" implies that the toolset is basic and general. Layer 3 includes tools that support specific applications and are needed either within a particular project or by an individual modeler [3]. For an historical overview of the SMDE project and a complete bibliography, see [5].

# 4 A Simulation Model Specification and Documentation Language

In his dissertation, Overstreet defines a formalism for simulation model specification in which the description of model behavior has several useful and desirable properties [22, p. 40]:

1. it is independent of traditional simulation world views,

2. a specification can be analyzed to identify natural components that measure complexity and identify potential problems with the specification,

3. a specification can be transformed to produce additional representations that conform to traditional simulation world views,
4. some aspects of a model can be left unspecified without hindering the analyses and transformations identified above, and
5. a model is defined in terms that do not prescribe any particular implementation techniques, such as the time flow mechanism.

The goal of this formalism, the condition specification (CS), is to provide a world-view-independent model representation that is sufficiently expressive to represent any model and formalized to facilitate automated diagnosis of the model representation. The CS is *not* proposed as a language with which a modeler directly works when creating a model – although it could be used as such. Rather, the CS is envisioned as a target form for analysis within a hierarchy of specifications for a given model as produced within an environment supporting simulation model development.

According to Overstreet any model specification must provide the following:

**Input specification** provides a description of the information the model receives from its environment.

**Output specification** provides a description of the information the environment receives from the model.

**Object definition set** provides a description of each model object and the attributes that are associated with it. (Note, Overstreet assumes an object-oriented view when specifying a model. While the identification of objects is not *essential* to model specification they are viewed as highly expedient in many cases. Nonetheless, attributes – or variables – *must* be defined in any model specification.)

**Indexing attribute** Commonly system time, all discrete event simulation models must contain an indexing attribute to allow the model to progress. The indexing attribute provides a partial ordering for the changes in state that occur during any execution of the simulation.

**Transition function** describes the initial state for the model (the values of all attributes of objects that exist at initiation and the initial value of system time), the conditions for termination, and a definition of the dynamic behavior of the model.

A condition specification of a model consists of two basic elements: a description of the communication interface for the model and a specification of model dynamics [22, p. 86]. The specification of model dynamics can be further divided into an object specification and a transition specification. The CS also identifies, but leaves syntactically undefined, a report specification which provides statistical reporting of simulation results. Details of the CS and examples of its use, as well as discussions of the forms of diagnostic assistance provided by the CS, can be found in [19, 20, 23]. The components of the CS – as originally defined in [22] – are briefly outlined here.

### 4.1 System interface specification

The system interface specification identifies input and output attributes by name, data type and communication type. (Overstreet assumes that the communication interface description can be derived from the internal dynamics of the model and can be system generated.) According to [23], every CS must have at least one output attribute.

### 4.2 Object specification

The object specification contains a list of all model objects and their defined attributes. The CS enforces typing for each attribute similar to those types used in Pascal: integer, real, boolean, or list values (enumerated typing). An additional type, time-based signal, is also provided.

### 4.3 Transition specification

A transition specification consists of a set of ordered pairs called "condition action pairs." Each pair includes a condition and an associated action. A condition is a boolean expression composed of model attributes and the CS sequencing primitives, WHEN ALARM and AFTER ALARM. Actions come in five classes: value change description, time sequencing action, object generation (or destruction), environment communication (input or output), or simulation run termination. The transition specification also permits the descriptions of functions (if any) used by a modeler to simplify the expressions of model behavior.

Condition action pairs (CAPs) with equivalent conditions are brought together to form action clusters. Action clusters (ACs) represent all actions which are to be taken in an implementation of a model whenever the associated condition is true.

Besides WHEN ALARM and AFTER ALARM, the CS provides sequencing primitives SET ALARM, and CANCEL which manipulate the values of attributes typed as time-based signals. Object generation is handled via primitives CREATE and DESTROY. The primitives INPUT and OUTPUT provide communication with the model environment. The conditions *initialization* and *termination* appear in every CS; initialization is true only at the start of a model instantiation (before the first change in value of system time). The expression for termination is model dependent and may be time-based, state-based, or mixed (contains both a time and a state condition).

### 4.4 Condition specification syntax

The syntax for the CS primitives is given in Figure 4. The CS as originally defined provides little support for many common specification notions (e.g. sets, queues) as well as offering no direct support for model statistical gathering functions (identifying, but not precisely defining, a report specification). These facilities were originally separated from the transition specification primarily because of their tendency to inhibit model analysis – indeed their inclusion would introduce many of the problems attendant to SPLs. Research is ongoing which extends the CS by providing for multiple levels – a *stratification* – of CS forms: a *baseline* form in which abstract data types (ADTs) are available and statistics are "automatically" gathered (in a manner similar to that provided by most simulation programming languages) according to an identification of the monitored attributes. (This form is designed to provide for succinct descriptions of model behavior). An *augmented* form may be automatically generated from the baseline specification by "in-line" expansion of the *macros* and *assumed* actions used within the baseline specification. (This form is the primary target for model analysis).

### 4.5 Support for model analysis

To motivate the discussion of model analysis within the condition specification, Overstreet [22] offers definitions for *model specification equivalence* noting that what is intuitively easy to understand is somewhat difficult to define mathematically in a succinct fashion. Intuitively, we accept that two model specifications are equivalent if and only if they can be used interchangeably. (Since Overstreet's concern is equivalence under transformations on model specifications, he does not address equivalence as it relates to the model objectives, that is that one model –

| *Name* | *Syntax* | *Function* |
|---|---|---|
| Value Change Description | object.attribute := *newValue* | Assign attribute values. |
| Set Alarm | SET ALARM( *alarmName, alarmTime* <, *argList*> ) | Schedule an alarm. |
| When Alarm | WHEN ALARM( *alarmName* <, *argList*> ) | Time sequencing condition. |
| After Alarm | AFTER ALARM( *alarmName* & *boolExpr* <, *argList*> ) | Time sequencing condition. |
| Cancel | CANCEL( *alarmName* ) | Cancel previously scheduled alarm. |
| Create | CREATE( *objectName*[*instanceId*] ) | Create new model object. |
| Destroy | DESTROY( *objectName*[*instanceId*] ) | Eliminate a model object. |
| Input | INPUT( *attribList* ) | Receive input from model environment. |
| Output | OUTPUT( *attribList* ) | Produce output to model environment. |
| Stop | STOP | Terminate simulation run. |

Figure 4: Condition Specification Syntax.

perhaps providing more information as a result of solving a larger (in scope) problem – may produce "more" information than another; if both models capture the study objectives, both may be used interchangeably. This issue is of some significance, in that, for reasons of enhanced software quality (lower maintenance costs, etc.) we desire models that *exactly* solve our problems; although we want these models to be readily extensible should the problems of interest change in some way.) Definitions for specification equivalence – as affected by specification transformations – can have either an analytic or statistical basis. Using a statistical approach, a probability statement with the equivalence of two (or more) specifications is generated based on the output of implementations of each. Alternatively, analytic methods can be used to determine if different specifications imply "equivalent" model actions under "equivalent" circumstances [22, p. 117].

### 4.5.1 Condition specification model decompositions

At its most elemental level, the condition specification produces condition action pairs. These CAPs may be aggregated into action clusters – groups of actions bound by a common condition. While CAPs and ACs form the basis of model representation in the CS, Overstreet defines several "decompositions" which provide varying levels of model description by couching the same information – which model conditions produce which model actions – in a variety of ways.

Among the transformations defined on a CS are equivalence-preserving transformations into each of the three traditional conceptual frameworks. Each of these transforms is based upon a graphical representation of a CS known as the action cluster incidence graph (ACIG) described in Section 4.5.2.

To produce an event scheduling orientation of a CS, the CAPs are organized around WHEN ALARMs – by generating a set of subgraphs from the ACIG which contain a single time-based action cluster and all the state-based action clusters reachable from it without passing through another time-based action cluster. Similarly, an activity scanning approach may be captured by creating subgraphs oriented around the state-based ACs in an ACIG, and a process interaction orientation may be generated by creating subgraphs of the ACIG that are actions as they relate to objects within the specification.

Providing a variety of descriptions of the same model enhances the model verification, validation and credibility assessment processes as well as portending the possibility of indicating efficient model implementations.

### 4.5.2 Action cluster incidence graphs

Once the ACs of a CS have been aggregated into CAPs the relationship among CAPs may be presented using an action cluster incidence graph (ACIG) or its matrix equivalent, the action cluster incidence matrix (ACIM).

An ACIG is a directed graph in which each node corresponds to an AC in the CS. If, during the course of any given implementation of the model, the actions in one action cluster, $AC_i$, cause the condition for another action cluster, $AC_j$, to become true (at either the same simulation time at which $AC_i$ is executed or at some future time by setting an alarm) then there is a directed arc from the node representing $AC_i$ to the node representing $AC_j$. By convention this arc is depicted as a dotted line if $AC_i$ sets an alarm that is used in the condition for $AC_j$. If the condition on $AC_j$ is a WHEN ALARM then $AC_j$ is referred to as a *time-based successor* of $AC_i$. If the condition on $AC_j$ is an AFTER ALARM then $AC_j$ is referred to as a *mixed successor* of $AC_i$. Otherwise $AC_j$ is referred to as a *state-based successor* of $AC_i$ and the arc is depicted as a solid line.

Graph representations can be of great assistance to model creation and comprehension, however when the graphs become large (both in numbers of nodes and arcs) the communication of model concepts suffers. To alleviate this problem, many graphically-oriented description languages provide a means of graph decomposition so that information is presented at varying levels, each of which contains an acceptable (relatively small) amount of information. No such decompositions are specifically defined for the ACIG – although reorganizing an ACIG to reflect a particular world view often simplifies the graph. Still, for very large models the graph may become unreadable and lose its value as a communicative device; for these models the ACIM is

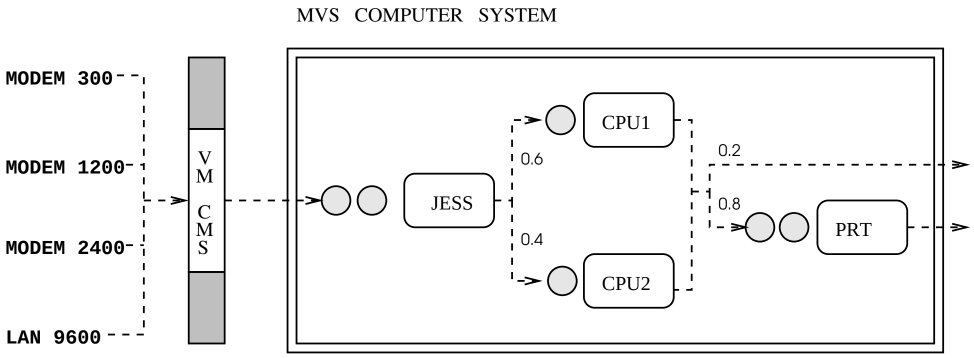


Figure 5: MVS System.

often the most useful representation.

For details of methods for the construction and simplification of ACIGs as well as the analyses provided by the ACIG, see [23, 25].

# 5 Example

In this section we present an example of the condition specification representation of a simple model. The development steps of the underlying conical methodology and the details of applying a model generator within the development environment are not illustrated. Our objective is to demonstrate a world-view-independent and implementation language-independent model representation and briefly describe how model implementation issues may be controlled and introduced in a well-defined manner so as not to interfere with model representation.

## 5.1 Multiple virtual storage model

Balci [4] describes a system in which a multiple virtual storage (MVS) batch computer operates with two central processing units (CPUs). Users submit batch programs to the MVS by using the *submit* command on an interactive virtual memory (VM) computer system running under the CMS operating system. As shown in Figure 5, the users of MVS via VM/CMS are classified into four categories: (1) users dialed in by using a modem with 300 baud rate, (2) users dialed in by using a modem with 1200 baud rate, (3) users dialed in by using a modem with 2400 baud rate, and (4) users connected to the local area network (LAN) with 9600 baud rate. Each

| **Type of User** | **Interarrival Times** | **Mean** |
|---|---|---|
| Modem 300 User | Exponential | 3200 Seconds |
| Modem 1200 User | Exponential | 640 Seconds |
| Modem 2400 User | Exponential | 1600 Seconds |
| LAN 9600 User | Exponential | 266.67 Seconds |

Table 1: Interarrival Times.

| **Facility** | **Processing Times** | **Mean** |
|---|---|---|
| JESS | Exponential | 112 Seconds |
| CPU1 | Exponential | 226.67 Seconds |
| CPU2 | Exponential | 300 Seconds |
| PRT | Exponential | 160 Seconds |

Table 2: Processing Times.

user develops the batch program on the VM/CMS computer system and submits it to the MVS for processing. Based on collected data, assume that the interarrival times of batch programs to the MVS with respect to each user type are determined to have an exponential probability distribution with corresponding means as shown in Table 1.

A batch program submitted first goes to the job entry subsystem (JES) of MVS. The JES scheduler (JESS) assigns the program to processor 1 (CPU1) with a probability of 0.6 or to processor 2 (CPU2) with a probability of 0.4. At the completion of program execution on a CPU, the program's output is sent to the user's virtual reader on the VM/CMS with a probability of 0.2 or to the printer (PRT) with a probability of 0.8. Assume that all queues in the MVS system are first-come-first-served and each facility processes one program at a time. The probability distribution of the processing times for programs by each facility is given in Table 2.

Assuming that the simulation model reaches steady state after 3,000 programs, simulate the system for 15,000 programs in steady state and construct confidence intervals for the following performance measures (known values given in parentheses):

1. Utilization of the JESS ($\rho_{JESS} = 0.70$).

2. Utilization of CPU1 ($\rho_{CPU1} = 0.85$).

3. Utilization of CPU2 ($\rho_{CPU2} = 0.75$).

4. Utilization of the PRT ($\rho_{PRT} = 0.80$).

5. Average time spent by a batch program in the MVS computer system ($W = 2400$ seconds).

6. Average number of batch programs in the MVS computer system ($L = 15$).

## 5.2 MVS object specification

Figure 6 presents the object specification for the MVS model. The model consists of six objects: the model object (mvs), jess, cpu1, cpu2, prt, and job, as well as queues for each of the facilities (in this example we include the use of queues as an extension to the CS for notational expedience). Two points can be made about this model definition: (1) The definition of jess, cpu1, cpu2, and prt as separate objects is done primarily for the benefit of this presentation. A single – multiply-instantiated – object, facility, could provide an equivalent definition. (2) The job object is required *only* as a result of the objectives of the study, i.e. report the average waiting time for a job in the system. Without this requirement, the facility queues could be represented by counters and the jobs themselves need not be uniquely identifiable. This is a good example of how the objectives of a simulation study dictate the model definition.

## 5.3 MVS transition specification

Figures 7 and 8 present the transition specification for the MVS model; the ACIG is pictured in Figure 9.

## 5.4 MVS function specification

Figure 10 presents the MVS function specification for a Pascal implementation. (Note that the provision for the definition of functions is merely a convenience for model description. In order to be useful for analysis, the function specification is incorporated into the transition specification when creating the *augmented specification* within the proposed extensions to the CS.)

| *Object* | *Attribute* | *CS Type* | *CM Type* | *Value(s)* |
|---|---|---|---|---|
| mvs | time_in_sys | pos. real | tti | |
| | num_jobs | pos. int. | sti | |
| | m300_iat | pos. real const. | pi | 3200.00 seconds |
| | m1200_iat | pos. real const. | pi | 640.00 seconds |
| | m2400_iat | pos. real const. | pi | 1600.00 seconds |
| | l9600_iat | pos. real const. | pi | 266.67 seconds |
| | cpu1_prob | pos. real const. | pi | 0.60 |
| | prt_prob | pos. real const. | pi | 0.80 |
| jess | status | ennumerated | sti | (busy,idle) |
| | arrival_m300 | time-based signal | tti | |
| | arrival_m1200 | time-based signal | tti | |
| | arrival_m2400 | time-based signal | tti | |
| | arrival_l9600 | time-based signal | tti | |
| | end_service | time-based signal | tti | |
| | mpt | pos. real const. | pi | 112.00 seconds |
| | route | enumerated | sti | (CPU1, CPU2, UNDEF) |
| cpu1 | status | ennumerated | sti | (busy,idle) |
| | end_service | time-based signal | tti | |
| | mpt | pos. real const. | pi | 226.67 seconds |
| | route | enumerated | sti | (PRT, EXIT, UNDEF) |
| cpu2 | status | ennumerated | sti | (busy,idle) |
| | end_service | time-based signal | tti | |
| | mpt | pos. real const. | pi | 300.00 seconds |
| | route | enumerated | sti | (PRT, EXIT, UNDEF) |
| prt | status | ennumerated | sti | (busy,idle) |
| | end_service | time-based signal | tti | |
| | mpt | pos. real const. | pi | 160.00 seconds |
| job | enter_time | pos. real const. | pi | |

| *Set* | *CS Type* | *CM Type* |
|---|---|---|
| jess_queue | queue of job objects | p-set |
| cpu1_queue | queue of job objects | p-set |
| cpu2_queue | queue of job objects | p-set |
| prt_queue | queue of job objects | p-set |

Figure 6: MVS Object Specification

```
{initialization}
Initialization:
  CREATE(jess)
  jess.status := idle
  SET ALARM(jess.arrival_m300,
          exp(mvs.m300_iat))
  SET ALARM(jess.arrival_m1200,
          exp(mvs.m1200_iat))
  SET ALARM(jess.arrival_m2400,
          exp(mvs.m2400_iat))
  SET ALARM(jess.arrival_l9600,
          exp(mvs.l9600_iat))
  jess.route := UNDEF
  CREATE(cpu1)
  cpu1.status := idle
  cpu1.route := UNDEF
  CREATE(cpu2)
  cpu2.status := idle
  cpu2.route := UNDEF
  CREATE(prt)
  prt.status := idle
  mvs.num_jobs := 0
  mvs.time_in_sys := 0.0
  CREATE(jess_queue)
  CREATE(cpu1_queue)
  CREATE(cpu2_queue)
  CREATE(prt_queue)
  system_time := 0.0

{termination}
mvs.num_jobs > 18000:
  STOP
  PRINT REPORT

{jess select route}
WHEN ALARM(jess.end_service):
  jess.route := jess_select_route()

{jess arrival m300}
WHEN ALARM(jess.arrival_m300):
  SET ALARM(jess.arrival_m300,
          exp(mvs.m300_iat))
  CREATE(job)
  job.enter_time := system_time
  ENQUEUE(job, jess_queue)

{jess arrival m1200}
WHEN ALARM(jess.arrival_m1200):
  SET ALARM(jess.arrival_m1200,
          exp(mvs.m1200_iat))
  CREATE(job)
  job.enter_time := system_time
  ENQUEUE(job, jess_queue)

{jess arrival m2400}
WHEN ALARM(jess.arrival_m2400):
  SET ALARM(jess.arrival_m2400,
          exp(mvs.m2400_iat))
  CREATE(job)
  job.enter_time := system_time
  ENQUEUE(job, jess_queue)

{jess arrival l9600}
WHEN ALARM(jess.arrival_l9600):
  SET ALARM(jess.arrival_l9600,
          exp(mvs.l9600_iat))
  CREATE(job)
  job.enter_time := system_time
  ENQUEUE(job, jess_queue)

{jess begin service}
jess.status = idle and NOT EMPTY(jess_queue):
  jess.status := busy
  SET ALARM(jess.end_service,exp(jess.mpt))
```

Figure 7: MVS Transition Specification (Part I)

```
{jess end service cpu1}
jess.route = CPU1:
    jess.status := idle
    jess.route := UNDEF
    job := DEQUEUE(jess_queue)
    ENQUEUE(job, cpu1_queue)

{jess end service cpu2}
jess.route = CPU2:
    jess.status := idle
    jess.route := UNDEF
    job := DEQUEUE(jess_queue)
    ENQUEUE(job, cpu2_queue)

{cpu1 begin service}
cpu1.status = idle and NOT EMPTY(cpu1_queue):
    cpu1.status := busy
    SET ALARM(cpu1.end_service, exp(cpu1.mpt))

{cpu1 select route}
WHEN ALARM(cpu1.end_service):
    cpu1.route := cpu_select_route()

{cpu1 end service prt}
cpu1.route = PRT:
    cpu1.status := idle
    cpu1.route := UNDEF
    job := DEQUEUE(cpu1_queue)
    ENQUEUE(job, prt_queue)

{cpu1 end service exit}
cpu1.route = EXIT:
    cpu1.status := idle
    cpu1.route := UNDEF
    job := DEQUEUE(cpu1_queue)
    mvs.time_in_sys := mvs.time_in_sys
            + (system_time - job.enter_time)
    DESTROY(job)
    mvs.num_jobs := mvs.num_jobs + 1

{cpu2 begin service}
cpu2.status = idle and NOT EMPTY(cpu2_queue):
    cpu2.status := busy
    SET ALARM(cpu2.end_service, exp(cpu2.mpt))

{cpu2 select route}
WHEN ALARM(cpu2.end_service):
    cpu2.route := cpu_select_route

{cpu2 end service prt}
cpu2.route = PRT:
    cpu2.status := idle
    cpu2.route := UNDEF
    job := DEQUEUE(cpu2_queue)
    ENQUEUE(job,prt_queue)

{cpu2 end service exit}
cpu2.route = EXIT:
    cpu2.status := idle
    cpu2.route := UNDEF
    job := DEQUEUE(cpu2_queue)
    mvs.time_in_sys := mvs.time_in_sys
            + (system_time - job.enter_time)
    DESTROY(job)
    mvs.num_jobs := mvs.num_jobs + 1

{prt begin service}
prt.status = idle and NOT EMPTY(prt_queue):
    prt.status := busy
    SET ALARM(prt.end_service, exp(prt.mpt))

{prt end service}
WHEN ALARM(prt.end_service):
    prt.status := idle
    job := DEQUEUE(prt_queue)
    mvs.time_in_sys := mvs.time_in_sys
            + (system_time - job.enter_time)
    DESTROY(job)
    mvs.num_jobs := mvs.num_jobs + 1
```

Figure 8: MVS Transition Specification (Part II)

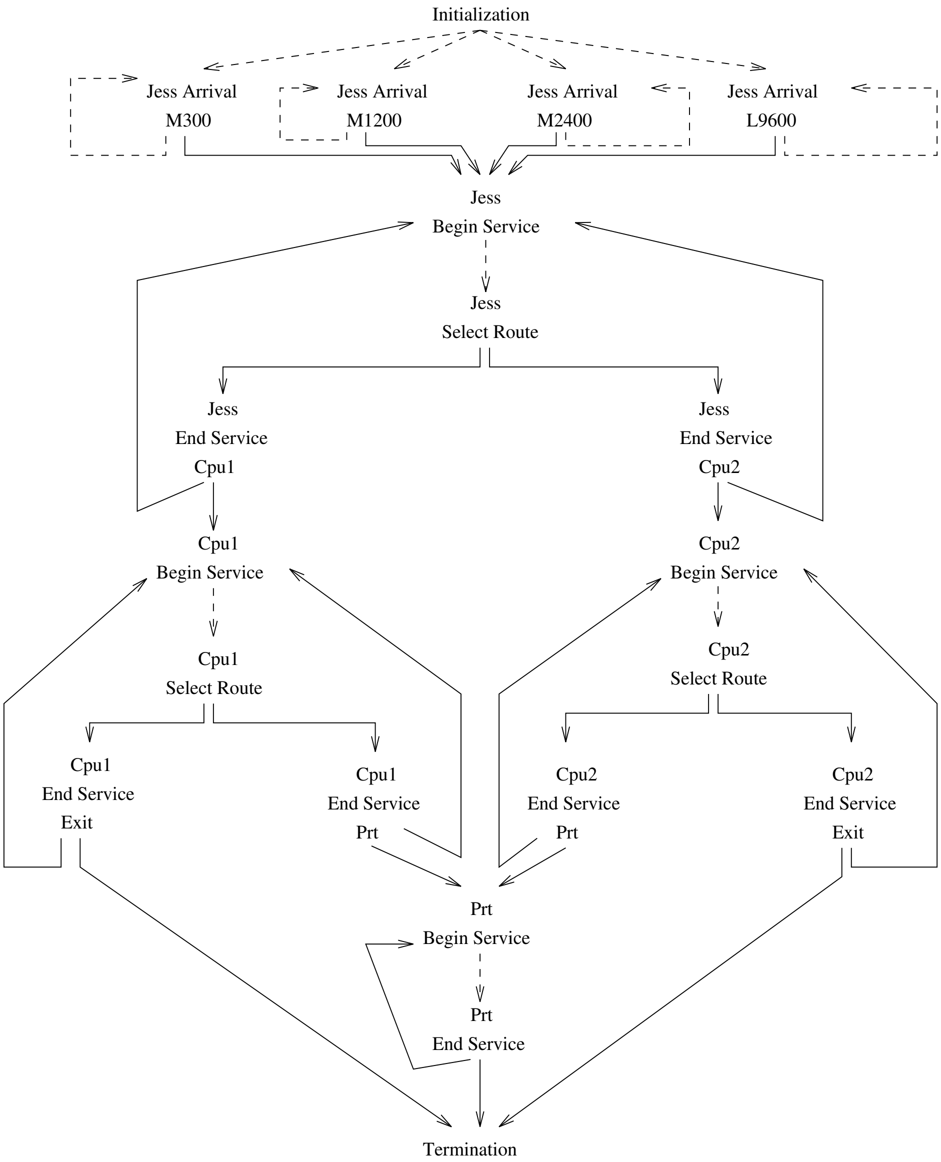


Figure 9: The Action Cluster Incidence Graph for MVS Model.

```
Function jess_select_route(var seed : integer) : integer;
var     x       : real;
begin
        x := random(seed);
        if x ≤ 0.6 then
                jess_select_route := CPU1
        else
                jess_select_route := CPU2
end (* jess_select_route *)
```

```
Function cpu_select_route(var seed : integer) : integer;
var     x       : real;
begin
        x := random(seed);
        if x ≤ 0.8 then
                cpu_select_route := PRT
        else
                cpu_select_route := EXIT
end (* cpu_select_route *)
```

```
Function random(var seed : integer) : real;
var     z       : real;
        quotr : real;
        quoti  : integer;
begin
        z := seed * 16807.0;
        quotr := z / 2147483647.00;
        quoti := trunc(quotr);
        z := z - quoti * 2147483647.00;
        seed := trunc(z);
        random := z / 2147483647.00;
end (* random *)
```

```
Function exp(mean : real; var seed : integer);
begin
        exp := - mean * ln(random(seed));
end (* exp *)
```

Figure 10: MVS Function Specification

## 5.5 MVS report specification

The CS report specification is defined to identify the performance measures of the simulation that are to be calculated and output upon termination of the simulation run.

Overstreet does not prescribe a form for the report specification, and separates it from the transition specification since the attributes and calculations required to gather statistics may clutter the description of model behavior, and thus be detrimental to the CS as a communicative model. For purposes of analysis, however, we would like *all* implementation "activity" to be present within the augmented transition specification.

Using a stratified approach, the CS report specification may take the form of a SIMSCRIPT-like declaration of variables and calculations in the baseline specification. This specification should provide the requisite information to construct the augmented transition specification. For sake of this example, we present a baseline CS report specification taking the form:

REPORT TIME IN STATE jess.status = busy OVER STEADY STATE DURATION AS
  "Utilization of JESS"

REPORT TIME IN STATE cpu1.status = busy OVER STEADY STATE DURATION AS
  "Utilization of CPU1"

REPORT TIME IN STATE cpu2.status = busy OVER STEADY STATE DURATION AS
  "Utilization of CPU2"

REPORT mvs.time_in_sys OVER mvs.num_jobs AS "Average time of job in system"

REPORT TIME WEIGHTED mvs.num_in_sys OVER STEADY STATE DURATION AS
  "Avg. number of jobs in system"

Notice that the transition specification given for the MVS example is essentially a baseline specification. The calculations to provide for the statistics gathering (with the exception of those needed to calculate the average time of a job in the system – done for purpose of illustration) have been omitted for clarity of presentation. (In fact, during the creation of CS examples this separation was done so often that the need for a stratified CS representation first became evident.) Note also that some means of describing the condition for the start of steady state must be provided in the specification. The developing CS extensions define an *experiment specification* that captures this information as well as other details needed to produce and experimental model implementation, such as the sources of model randomness and initial seeds for the requisite random variate generation, and so on.

## 6 From Specification to Implementation

This paper begins with a reference to a seldom challenged adage about the nature of problem solving. We conclude the paper by recalling another often used but seldom challenged adage that to be unfamiliar with history is to be doomed to repeat it. We consider this sage advice in the context of the evolution of computer software.

During the dawn of the age of digital computers, software was typically inelegant – often "jury-rigged" – code written in the language of a particular machine. In many cases the software was written by the machine designers themselves. Programming – such as it was – seemed as much witchcraft as anything else.

The unprecedented growth in integrated circuit technology that took place in the 1970's, where orders-of-magnitude improvements could be expected every six months to a year resulted in what is commonly referred to as the *software crisis.* Machines were becoming more and more powerful, portending the capability of solving problems of an ever-increasing scale. Unfortunately, as problems became more complex so did the software required to solve them. And techniques for developing large programs were not keeping pace with the hardware advances.

This crisis served as the progenitor of a field within computer science dedicated to identifying and understanding the *laws* that govern software as it is developed and used – chiefly in an attempt to bring predictability to a process that until that time often seemed chaotic. The new field came to be termed *software engineering.* (Initially it was believed that the construction and use of software could be *engineered* in much the same way as a bridge is engineered. However, the passage of time has indicated that perhaps this is not now, nor is ever likely to be, the case. And the *engineering* of software is accepted as an established misnomer.)

One of the earliest contributions of this de facto search for a methodological system for software construction was the enunciation by Edsger Dijkstra [10] of the principle of "separation of concerns" as a central factor in the development of large-scale software systems. Frederick Brooks [7] describes a similar tenet in his call for the need to remove the "accidental" complexities of software development from those "essential" complexities. The primary motivation of each of these precepts is the desire to increase the probability of producing error-free software. Correctness became recognized as an important feature of *software quality*, and these principles dictated that the likelihood of producing correct software hinged in large part on the ability of the software designer to separate the description of *what* must be done from the details of *how* it must be done. In other words, these canons call for representational forms devoid of the kind of implementation details that accompany programming language representations.

Other software quality objectives, as well, derive largely from the model representation process, among them: reliability, testability, reusability, adaptability and maintainability. All can be greatly enhanced or impaired by model representation. And since the cost of maintenance has been demonstrated to exceed seventy percent of the life cycle cost of software [21], any approach which fails to adequately support model representation would almost certainly prove cost-ineffective.

The lessons of this history would seem to be self evident. But consider, for instance, parallel (or distributed) discrete event simulation – the field seeking to provide mechanisms by which to do discrete event simulation on multiprocessors or distributed networks. The focus of a great deal of the research in this area falls on the representational requirements that permit maximum exploitation of parallelism. Often this implies a *formulation* of the model based upon a proper distribution of data in order to facilitate an acceptable balance of the computational load as well as a minimization of bottlenecks. Certainly, a description of the behavior of the system to be modeled does not require any of these details to be resolved. To advocate forcing a modeler *when specifying a model* to worry about whether or not a particular object will act as a performance bottleneck is tantamount to calling for a return to the days when a programmer had to explicitly consider register allocation and memory address mappings. The cost of this would be incurred in software quality: a second software crisis.

That science must progress in this fashion can be argued. Must small incremental solutions to localized problems provide the best path to a generalized solution of the larger issues? Perhaps. However, even if this is so, it is incumbent upon those taking the small steps to have a recognition – and at least seek to attain an understanding – of the global objective. Without this *big picture* in mind, one may easily, to refer to yet another adage, lose sight of the forest for the trees.

# 7 Conclusions

Model representation is a critical part of any model-centered problem solving effort. The quality of the solution is highly correlated with the quality of the model. The quality of the model is a function of the methodology and tools used to develop it. Too often, in software-related modeling, a program-centered approach to model generation is accepted as adequate, when in fact history has indicated otherwise.

Model formulation must first be founded in the context of a methodological approach to model development and usage, and second guided by the paradigm of the language (or languages) of the model representation. When these languages are implementation-based, or the methodologies prescribe a role to model representation other than that of strictly enunciating one or more aspects of system behavior, the model representation process becomes unduly complex and the model representations suffer accordingly.

In this paper we have discussed the role of model representation in discrete event simulation as viewed in the context of a life cycle model for a simulation study. A methodology and environment supporting the life cycle – and explicitly recognizing the key role of model representation within the life cycle – are briefly described. We describe the requirements for a simulation model specification and documentation language and present one language, the condition specification, as an example fulfilling many of these requirements. Given the growing diversity of uses for discrete event simulation models, from real-time decision support to interactive visualization in a training environment, the ability of any one representation form to suffice for all uses is increasingly unlikely. The *next generation* simulation model specification and documentation language is likely to be a *set* of languages and target forms: one for a high-level model description, one defining the system for animation, one providing the basis for model analysis, etc. This multiplicity of model representation only accentuates the need for methodological and environment support for the model development process, and further ensconces the importance of model representation within the life cycle.

## References


[1] 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 (TR CS-86-24) Systems Research Center and Dept. of Computer Science, Virginia Tech, Blacksburg, VA, September.

[2] Barger, L.F. (1986). "The Model Generator: A Tool for Simulation Model Definition, Specification, and Documentation," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, August.

[3] Balci, O. (1986). "Requirements for Model Development Environments," *Computers and Operations Research*, **13**(1), pp. 53-67.

[4] Balci, O. (1988). "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages," Technical Report SRC-88-009 (TR-88-21), Systems Research Center and Dept. of Computer Science, Virginia Tech, Blacksburg, VA, August.

[5] Balci, O., and Nance, R.E. (1992). "The Simulation Model Development Environment: An Overview," Technical Report SRC-92-004 (TR-92-32), Systems Research Center and Dept. of Computer Science, Virginia Tech, Blacksburg, VA, June.

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

[7] Brooks, Jr,. F.P. (1975). *The Mythical Man Month,* Addison-Wesley Publishing Company, Reading, MA.

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

[9] Derrick, E.J. (1988). "Conceptual Frameworks for Discrete Event Simulation Modeling," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, August.

[10] Dijkstra, E. (1972). "Notes on Structured Programming," *Structured Programming,* Dahl, O-J, Dijkstra, E., and Hoare, C.A.R., eds., Academic Press.

[11] Kiviat, P.J. (1967). "Digital Computer Simulation: Modeling Concepts," RAND Corp. Memorandum RM-5378-PR, Santa Monica, CA, January.

[12] Lackner, M.R. (1962). "Toward a General Simulation Capability," In: *Proceedings of the SJCC,* AFIPS Press, pp. 1-14.

[13] Nance, R.E. (1971). "On Time Flow Mechanisms for Discrete Event Simulations," *Management Science,* **18**(1), pp. 59-93, September.

[14] 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. (1981). "The Time and State Relationships in Simulation Modeling," *Communications of the ACM,* **24**(4), pp. 173-179, April.

[16] Nance, R.E. (1983). "A Tutorial View of Simulation Model Development," In: *Proceedings of the 1983 Winter Simulation Conference,* pp. 325-331, Arlington, VA, December 12-14.

[17] Nance, R. E. (1993). "A History of Discrete Event Simulation Programming Languages," In: *Proceedings of the Second ACM SIGPLAN History of Programming Languages Conference,* Cambridge, MA, April 20-23, Reprinted in *ACM SIGPLAN Notices,* **28**(3), pp. 149-175.

[18] Nance, R.E., and Balci, O. (1987). "Simulation Model Management Objectives and Requirements," *Systems and Control Encyclopedia: Theory, Technology, Applications.* M.G. Singh, Ed., Pergamon Press, Oxford, pp. 4328-4333.

[19] Nance, R.E., and Overstreet, C.M. (1987). "Exploring the Forms of Model Diagnosis in a Simulation Support Environment," In: *Proceedings of the 1987 Winter Simulation Conference,* pp. 590-596, Atlanta, GA, December 14-16.

[20] Nance, R.E., and Overstreet C.M. (1988). "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," *Transactions of the Society for Computer Simulation,* The Society for Computer Simulation, **4**(1), pp. 33-57, January.

[21] Neighbors, J.M. (1984). "The DRACO Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering,* **SE-10**(9), pp. 564-574, September.

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

[23] 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, February.

[24] Page, E.H. (1990). "Model Generators: Prototyping Simulation Model Definition, Specification, and Documentation Under the Conical Methodology," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, August.

[25] Puthoff, F.A. (1991). "The Model Analyzer: Prototyping the Diagnosis of Discrete-Event Simulation Model Specification," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, September.

[26] Watson, C.E. (1976). "The Problems of Problem Solving," *Business Horizons,* **19**(4), pp. 88-94, April.

[27] Wegner, P. (1987). "The Object-Oriented Classification Paradigm," *Research Directions in Object-Oriented Programming,* Wegner, P. and Shriver, B. (Eds), MIT Press, Cambridge, MA.

[28] Zeigler, B.P. (1976). *Theory of Modeling and Simulation,* John Wiley and Sons, New York.