A HISTORY OF DISCRETE EVENT SIMULATION PROGRAMMING LANGUAGES

Richard E. Nance
Director, Systems Research Center
Professor of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061-0251

ABSTRACT
The history of simulation programming languages is organized as a progression in periods of similar developments. The five periods, spanning 1955–1986, are labeled: The Period of Search (1955–1960); The Advent (1961–1965); The Formative Period (1966–1970); The Expansion Period (1971–1978); and The Period of Consolidation and Regeneration (1979–1986). The focus is on recognizing the people and places that have made important contributions in addition to the nature of the contribution. A balance between comprehensive and in-depth treatment has been reached by providing more detailed description of those languages that have or have had major use. Over 30 languages are mentioned, and numerous variations are described. A concluding summary notes the concepts and techniques either originating with simulation programming languages or given significant visibility by them.

CONTENTS
8.1 Introduction
8.2 The Period of Search (1955–1960)
8.3 The Advent (1961–1965)
8.4 The Formative Period (1966–1970)
8.5 The Expansion Period (1971–1978)
8.6 Consolidation and Regeneration (1979–1986)
8.7 Concluding Summary
References
8.1 INTRODUCTION
This introductory section is intended to explain the different types of computer simulation and to identify the key issues in simulation programming languages (SPLs). The approach to this survey is the subject of a concluding subsection.

8.1.1 Computer Simulation
Analog (or analogue) simulation, that is, simulation on analog computers, is not addressed in this survey because no major "language" or language issues are associated with this early form. However, analog simulation was much in evidence during the 1950s and earlier. Digital simulation, that is, simulation on digital computers, is considered synonymous with "computer simulation" for this history of SPLs.

8.1.1.1 Taxonomy and Terminology
Computer simulation is an application domain of programming languages that permits further division into three partitions:

1. discrete event simulation,
2. continuous simulation, and
3. Monte Carlo simulation.

Discrete event simulation utilizes a mathematical/logical model of a physical system that portrays state changes at precise points in simulated time. Both the nature of the state change and the time at which the change occurs mandate precise description. Customers waiting for service, the management of parts inventories, or military combat are typical application domains for discrete event simulation.

Continuous simulation uses equational models, often of physical systems, which do not portray precise time and state relationships that result in discontinuities. The objectives of studies using such models do not require the explicit representation of state and time relationships. Examples of such systems are found in ecological modeling, ballistic reentry, or large-scale economic modeling.

Monte Carlo simulation, the name given by John von Neumann and Stanislaw M. Ulam [Morgen-thaler 1961, p. 368] to reflect its gambling similarity, utilizes models of uncertainty where representation of time is unnecessary. The term is originally attributed to "a situation in which a difficult nonprobabilistic problem is solved through the invention of a stochastic process that satisfies the relations of the deterministic problem" [Morgenenthaler 1961]. A more recent characterization is that Monte Carlo is "the method of repetitive trials" [Shreider 1966, p.1]. Typical of Monte Carlo simulation is the approximation of a definite integral by circumscribing the region with a known geometric shape, then generating random points to estimate the area of the region through the proportion of points falling within the region boundaries.

Simulation as a problem-solving technique precedes the appearance of digital computers by many years. However, the emergence of digital technology exerted an influence far more than adding the term "computer" as an adjectival descriptor of this method of problem solving. The oppressive manual computational burden was now off-loaded to a much faster, more tolerant processor—the digital computer.

Three related forms of simulation are commonly noted in the literature. Combined simulation refers generally to a model that has both discrete event and continuous components (see [Law 1991, p. 112]). Typically, a discrete event submodel runs within an encapsulating continuous model. Hybrid simulation refers to the use of an analytical submodel within a discrete event model (see [Shantikumar 1983]).
Finally, *gaming* or *computer gaming* can have discrete event, continuous, and/or Monte Carlo modeling components.

### 8.1.1.2 Language Requirements for Discrete Event Simulation

In her description of programming languages, Sammet [1969, p. 650] justifies the categorization of simulation languages as a specialization warranting limited description with the statement that, “their usage is unique and presently does not appear to represent or supply much carry-over into other fields.” The impact of the object-oriented paradigm, and the subsequent clamor over object-oriented programming languages, first represented by SIMULA, would appear in hindsight to contradict this opinion, which accurately represented the attitude of the programming language research community at that time.

A simulation language must provide a model representation that permits analysis of execution behavior. This provision entails six requirements:

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 created, deleted, and manipulated 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. timing executive or a time flow mechanism, to provide an explicit representation of time.

Every simulation programming language provides these components to some degree.

Many simulation applications are undertaken in general purpose languages. Strongly influenced by this fact, simulation “languages” have taken three forms: package, preprocessor, and conventional programming language. A package is a set of routines in language X that can be called or invoked to meet the six requirements listed previously. The user (programmer) might develop additional routines in X to provide needed capabilities or a tailored treatment. Although such packages are not truly languages, some have had major influence, perhaps leading to descendants that take one of the other two forms. For example, the evolution of GASP, a package, produced SLAM, a preprocessor. All three “language” forms are included because examples of each have played significant roles in shaping the simulation scene.

### 8.1.1.3 Conceptual Frameworks for Discrete Event Modeling

Very early, Lackner [1962, p. 3] noted the importance of a modeling perspective as a “WELTAN-SICHT” or world view that “must be implicitly established to permit the construction of a simulation language.” In a subsequent work, he attempted to lay the groundwork for a theory of discrete event simulation and used the differing world views to categorize SPLs [Lackner 1964]. Lackner’s categorization differentiates between event-oriented and activity-oriented SPLs. Kiviat [1967; 1969] expanded the categories to identify three world views: event-oriented, activity-oriented, and process-oriented.
Fishman [1973] helped to clarify the distinction among the categories, which he labeled “event scheduling,” “activity scan,” and “process interaction.” These labels have become more or less the conventions in the literature. A recent thesis by Derrick [1988] expanded the three to 13, doing so in a manner that encompassed more than SPL discrimination.

The differentiation among world views characterized by languages is captured best using the concept of locality [Overstreet 1986, p. 171]:

*Event scheduling* provides locality of time: each event routine in a model specification describes related actions that may all occur in a single instant.

*Activity scanning* provides locality of state: each activity routine in a model specification describes all actions that must occur due to the model assuming a particular state (that is, due to a particular condition becoming true).

*Process interaction* provides locality of object: each process routine in a model specification describes the entire action sequence of a particular model object.

### 8.1.2 Historical Approach

Although fundamental in distinguishing the modeling differences among SPLs, the world views do not suggest a basis for historical description for the following reasons:

1. the origins and initial development of the language(s) promoting a particular view took place concurrently without a clear progression of conceptual convergence, and
2. remarkably parallel development patterns are evident throughout the major SPLs, irrespective of world view ties.

However, the extent and range of activity in SPLs mandates an organizational thesis, and a chronological presentation is used.

A chronological depiction of SPL development could proceed in several ways. One could provide a vertical trace of developments within each SPL. The tack taken in this paper is more of a horizontal cut across periods of time, which are noted in the genealogical tree of SPLs, shown in Figure 8.1:

- 1955–1960: The Period of Search
- 1971–1978: The Expansion Period
- 1979–1986: The Period of Consolidation and Regeneration

Each major language originating in a period is described in terms of its background and rationale for language content. Description of that language in subsequent time periods is limited to changes in that language, particularly those that mark the designated time period. Descriptions of GPSS and SIMULA are limited also; since both languages are treated extensively in the first conference devoted to programming language history (see [Wexelblatt 1981]). No attempt is made to describe language developments since 1986; sufficient time has yet to elapse to permit a historical perspective.

Figure 8.1 is organized vertically to distinguish languages according to world view. This categorization cannot be precise, for the language developer in most instances was unaware of the distinguishing characteristic. Crookes [1982, p. 1] places the number of SPLs as 137, although giving
FIGURE 8.1
The Genealogical Tree for Simulation Programming Languages.
no source, and attributes this overpopulation to a lack of standardization in general purpose computer languages. More likely, the excess in SPLs can be attributed to a combination of root causes:

1. perceived differences among application domains, requiring language features not provided in extant SPLs;
2. lack of awareness of SPLs or the reliance on the general purpose language "crutch" leading to package development;
3. educational use of a well known language in which components are provided so that learning time is reduced and the cost of an additional translator is avoided; and
4. competition among language vendors to exploit particular features or selected application domains.

The degree to which each of these causes has proved influential is quite dependent on time. The first and second tend to mark the early periods; the fourth, the later periods.

One further point concerns the unfortunate but all too common instances of incomplete references, primarily missing dates. In the attempt to provide as much historical information as possible, dates are given if suggested by other sources, added notes (time stamps), or personal recollection. In those cases where the data is uncertain, a "?” suffix is placed in the citation, for example, [Meyerhoff 1968?] indicates that the recollection of Paul Roth is that this undated manual was completed in 1968 [Roth 1992].

8.2 THE PERIOD OF SEARCH (1955–1960)

The title of this subsection is derived from the efforts during this period to discover not only concepts of model representation but to facilitate the representational needs in simulation modeling. Simulation languages came into being with the recognition of the power of the technique in solving problems that proved intractable to closed-form mathematical solution. The most prominent examples of such problems are the early manufacturing simulations by a group at General Electric; Jackson, Nelson, and Rowe at UCLA; and Baker and Dzielsinski cited in Conway [1967, p. 219].

K.D. Tocher is generally credited with the first simulator described in a 1960 publication jointly with D.G. Owen [Tocher 1960]. The simulator, later called GSP (General Simulation Program), introduced the three-phase method of timing control generally adopted by activity scan languages. Significantly, Tocher characterized his work as "searching for a simulation structure," rather than merely providing a "simulation language," [Tocher 1960, p.51]. Excerpts from the GSP example given in the appendix of the 1960 paper are shown in Figure 8.2. Although lacking the format regularity of assembly language programs, the syntactic structure of GSP seems to fall far short of the descriptor "automatic programming of simulations," [Tocher 1960, p.51]. However, this seemingly inflated claim was typically applied to languages other than basic machine coding, including assemblers (see [Rosen 1967] for examples).

In the United States interest in simulation was emerging rapidly in the late 1950s; however, the recognition of concepts and the assistance of languages in dealing with them was less recognized. Gordon [1981] describes his own efforts prior to 1960 as the writing of simulation programs for message switching systems. The Sequence Diagram Simulator, described in a conference in 1960, indicates some appreciation of language support for the application task [Gordon 1981, p. 405].

The treatment given to computer simulation in the 1961 issue of Progress in Operations Research [Ackoff 1961] demonstrates the perceived importance of the technique. However, the issue of manual
FIGURE 8.2
Excerpts from a GSP Program.

```
EXAMPLE FOR I.P.O.R.S. PAPER
Title of Job.
B1-5 B1 0 0 0 B3 for time-dependent
T1-5 1 0 0 0 1440
U5 1440
Z

PARAMETERS
R-U11 +1.62 +1000 +.576 +.4 Parameters
L
R-W12 +.529 +6913 +75859 +129459 +5612 +3095 for routines
L
R-M13 +.711 +228121 +98655 +260848 +9995 +9162 SAMPLE and
L defining
R-M14 +.293 +120440 +30873 +45456 +8360 +2879 etc.
L
R-M15 +.801 +76869 +18934 +176332 +7907 +2069
L
R-M16 +.417 +228218 +98632 +228080 +9993 +9161
L
R-M17 +.516 +28675 +61077 +66404 +3897 +3295
L
Z

DURATION 14400 Simulation to run for 10 days at a
time.

TRANSLATE (End of Initial Conditions.)

C1 PROCESSING Activity C1
K = 0
n/2, A(n + 2), U = 0.

Q(n + 2) = XYZ T11
Q(n + 2) = SAMPLE R(nn) + Q(n + 2) if there is a non-zero queue, and an
n/2.

v = 3 up-date idle time, start processing,

C2 UNLOADING Activity C2
K = 0
n/2, A(n + 2), U = 1.

Q(n + 2) = XYZ T11
Q(n + 2) = SAMPLE R(nn) + Q(n + 2) if there is a non-zero queue, and an
n/2.

B1 ARRIVALS Activity B1
v = NGEKP R11/S
pl = 1, D
x = 1, x, M,
W = X
K
B2 TRANSFER TO STOCK Activity B2
v = SAMPLE R(14 + Qn)
Mn = 0, D
H = P + U2 + U3 + X
K, STOP
B3 REPORT RESULTS Activity B3
PRINT.C W1
PRINT.A V2
PRINT.E W2
PRINT.A V3
PRINT.E W3
PRINT.A Wn
PRINT.A Wn
v = U2

SUBROUTINE XYZ

XYZ 5.6 700
6 100
0 0717
0.1 1037
0 600
5.1 162
1.4 600
5.1 600
100
0 000.
0 +0
0 +0
10 600
5.1 610
5.6 710
9.0 7007
1.6 720
0 450
P

Here, the text of the subroutine is supplied in
Pegasus order-code. It will be incorporated
in the specification of the model.
(There is a pair of output routines, the
quality '3' or the quality 'zero.' The former
appears with probability given by element
011. A pseudo-random number stored in T11
is used.)
```
versus computer execution is still a matter of consideration [Morgenthaler 1961, pp. 406-407], and little consideration of language needs is indicated.

8.3 THE ADVENT (1961–1965)

The period of 1961–1965 marks the appearance of the forerunners of the major simulation programming languages currently in use. This claim is documented to some extent in the genealogical tree for SPLs (Figure 8.1).

The historical development of GPSS and SIMULA are described in detail in Wexelblatt [1981]. Herein, each is described sufficiently to furnish a trace of the evolution of the language through the subsequent time periods. More detailed description is provided for other languages emerging during this period, although none can be described to the extent permitted in the first conference.

8.3.1 GPSS

Beginning with the Gordon simulator in 1960, the General Purpose System Simulator (GPSS) was developed on various IBM computers, for example, 704, 709, and 7090, during 1960–1961. With the later name change to the General Purpose Simulation System, GPSS developments continued under the version labeled GPSS II, which transitioned from table-oriented to list processing techniques. The GPSS III version released in 1965 was made available primarily for the large IBM systems (7090/94 and 7040/44) [IBM 1965]. A major issue was the lack of compatibility between GPSS II and GPSS III.

GPSS has the distinction of being the most popular SPL. Uses and users of the language outnumbered all others during the early years (prior to 1975). No doubt, this popularity was due in part to the position of IBM as the premier marketing force in the computing industry, and GPSS during this era of unbundled software was the discrete event SPL for IBM customers. That fact alone, however, does not explain the appeal of GPSS. The language readily gained proponents in the educational community because of the speed with which noncomputer-oriented students could construct a model of a queueing system and obtain numerical results. Originating in the problem domain of communication systems, the GPSS block semantics were ideally suited for queueing models (although logic switches might have puzzled many business students).

The importance of graphical output for on-line validation was demonstrated in 1965 using an IBM 2250 interactive display terminal tied to a GPSS model [Reitman 1992]. Boeing (Huntsville) was the first to employ this device to enable the observation of model behavior and the interruption and restart during execution.

8.3.2 SIMULA I

The selection of GPSS and SIMULA by the program committee for the first conference on the History of Programming Languages (HOPL I) is notable in that the two represent opposite poles in several measurement scales that could be proposed for languages. Within the simulation community in the United States (US), GPSS was viewed as easy to learn, user friendly (with the symbolic interface), highly used, limited (in model size and complexity), inflexible (especially so prior to the introduction of the HELP block), and expensive to run (the interpretive implementation). On the contrary, SIMULA was considered difficult to learn (lack of ALGOL knowledge in the US), lacking in its human interface (with input/output being an implementation decision following ALGOL 60), little used (in the US), but conceptually innovative and broadly applicable. The discovery of SIMULA by the programming
language research community in the 1970s was probably the influencing factor in its selection for HOPL I rather than the regard held for the language by simulation practitioners (notwithstanding its excellent reputation among a small cadre of simulation researchers in the US and a larger group in Europe).

Originating as an idea in the spring of 1961, SIMULA I represented the work primarily of Ole-Johan Dahl and Kristen Nygaard. The language was developed for the Univac 1107 as an extension of Univac's ALGOL 60 compiler. Nygaard and Dahl [1981] described the development of the language during this period as progressing through four main stages: (1) a discrete event network concept, (2) basing of the language on Algol 60, (3) modification and extensions of the Univac ALGOL 60 compiler, and the introduction of the "process concept," and (4) the implementation of the SIMULA I compiler.

The technical contributions of SIMULA are impressive, almost awesome. Dahl and Nygaard, in their attempt to create a language where objects in the real world could be accurately and naturally described, introduced conceptual advances that would be heralded as a paradigmatic shift almost two decades later. Almost lost in the clamor over the implementation of abstract data types, the class concept, inheritance, the coroutine concept, and quasi-parallel execution were the solution of significant problems in the extension of ALGOL 60. The creation, deletion, and set manipulation operations on objects are but one example. A second is that the sharing of attribute values by interacting processes required a means for breaking down the scoping restrictions of ALGOL 60. During the luncheon with language presenters at HOPL I in 1978, Nygaard remarked that only those who had programmed simulations really understood the power of the language (see [Wexelblat 1981, p. 485]).

8.3.3 SIMSCRIPT

SIMSCRIPT, in its original form was an event scheduling SPL. Two ancestors contributing to SIMSCRIPT are identified in the Preface to Markowitz [1963]: GEMS (General Electric Manufacturing Simulator) developed by Markowitz at the General Electric Manufacturing Services and SPS-1 developed at RAND. Morton Allen was identified as a major contributor to GEMS, and contributors to SPS-1 included Jack Little of RAND and Richard W. Conway of Cornell University.

8.3.3.1 Background

The RAND Corporation developed SIMSCRIPT under the auspices of the U.S. Air Force. The IBM 709/7090 computer was the target machine, and copies of SIMSCRIPT were distributed through SHARE (the IBM user's group). Harry Markowitz, whose broad contributions in several areas were to earn him a Nobel Prize, is generally attributed with the major design, and Bernard Hausner was the sole programmer, actually for both SPS-1 and the SIMSCRIPT translators. Herbert Karr authored the SIMSCRIPT manual [Markowitz 1963, p. iii].

SIMSCRIPT was considered a general programming system that could treat nonsimulation problems, but the major purpose of the language was to reduce the model and program development times that were common in large efforts. Users were intended to be noncomputing experts, and the descriptive medium for model development was a set of forms: (1) a SIMSCRIPT definition form in which variables, sets, and functions are declared, (2) an initialization form to define the initial system state, and (3) a report generation layout, for prescribing the format and content of simulation output. Accompanying the forms in the model definition is a routine for each endogenous and exogenous event and for any other decisions that prescribe the model logic. The applications context was machine or job shop scheduling and the initial examples reflect this focus.
8.3.3.2 Rationale for Language Content

Influencing Factors  The first version of SIMSCRIPT provided a set of commands that included FORTRAN statements as a subset. The syntax of the language and program organization were influenced considerably by FORTRAN. Basically, the language was transformed by a preprocessor into FORTRAN before compilation into the executable object program version. Insertion of a SIMULATION or NON-SIMULATION card in the compile deck differentiated between the intended uses, permitting omission of the events list and timing routines for nonsimulation purposes. A control routine designated as MAIN replaced the timing routine.

SIMSCRIPT 1.5, a version proprietary to CACI [Karr 1965], is described by Markowitz [1979, pp. 27-28] as a "slightly cleaner version of SIMSCRIPT I" that did not appear much different externally. However, the SIMSCRIPT 1.5 translator did not generate FORTRAN statements but produced assembly language. SIMSCRIPT 1.5 is also described as germinating many of the ideas that later appeared in SIMSCRIPT II.

Language Design and Definition  SIMSCRIPT I (and SIMSCRIPT 1.5) describe the world in terms of entities, attributes, and sets. Entities can be either permanent (available throughout the simulation duration) or temporary (created and destroyed during the simulation). Attributes define the properties of entities, and are distinguished by their values. Sets provide a convenience to the modeler for description of a class of entities (permanent entities), and set ownership and membership are defined relationships among entities.

The model structure is described in terms of a listing of events to prescribe the dynamic relationships, coupled with the entity and attribute definitions that provide a static description. Figure 8.3 taken from Markowitz [1963, p. 21] shows the declaration of exogenous and endogenous events for a job shop model. Figure 8.4 shows one exogenous input-an event with the name ORDRIN, also taken from Markowitz [1963, p. 22]. The event declarations create event notices that enable the timing executive to manipulate the events list, advancing time to the next imminent event. System-defined variables and functions such as TIME, RANDM, and PAGE are accessible by the modeler.

Nontechnical Influences  Shortly after SIMSCRIPT I appeared, Karr left RAND to form Consolidated Analysis Centers, Incorporated (CACI). CACI was the major force behind SIMSCRIPT I.5, and implementations were done by Claude M. Delfosse, Glen Johnson, and Henry Kleine (see the Preface to [CACI 1983]).

Referring to the language requirements described in section 8.1, Figure 8.5 gives examples of each of the six language requirements. The terminology used in the middle column of Figure 8.5 follows that of the first version of SIMSCRIPT. Upper case in the examples shows the required syntax.

8.3.4 CSL

Control and Simulation Language (CSL) was a joint venture of Esso Petroleum Company, Ltd. and IBM United Kingdom, Ltd. The purpose of the language was to aid in the solution of complex logical and decision making problems in the control of industrial and commercial undertakings [Esso 1963].

8.3.4.1 Background

[Buxton 1962] describes the simulation capabilities of CSL as based on the General Simulation Program [Tocher 1960] and Montecode [Kelly 1962]. The language is described as experimental in
FIGURE 8.3
Events List for Example—Job Shop Simulation in SIMSCRIPT.

<table>
<thead>
<tr>
<th>STATEMENT NUMBER</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EVENTS</td>
</tr>
<tr>
<td>2</td>
<td>EXOGENOUS</td>
</tr>
<tr>
<td>5</td>
<td>ORDRIN (1)</td>
</tr>
<tr>
<td>6</td>
<td>ANALYZ (2)</td>
</tr>
<tr>
<td>7</td>
<td>ENDSIM (3)</td>
</tr>
</tbody>
</table>

FIGURE 8.4
Exogenous Event Routine Describing the Receipt of an Order.

<table>
<thead>
<tr>
<th>STATEMENT NUMBER</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EXOGENOUS EVENT ORDRIN</td>
</tr>
<tr>
<td>2</td>
<td>SAVE EVENT CARD</td>
</tr>
<tr>
<td>5</td>
<td>CREATE ORDER</td>
</tr>
<tr>
<td>6</td>
<td>LET DATE(ORDER) = TIME</td>
</tr>
<tr>
<td>7</td>
<td>READ N</td>
</tr>
<tr>
<td>72</td>
<td>FORMAT (14)</td>
</tr>
<tr>
<td></td>
<td>DO TO 10, FOR I = (1) (N)</td>
</tr>
<tr>
<td></td>
<td>CREATE DESTN</td>
</tr>
<tr>
<td></td>
<td>READ MGDST(DESTN), PTIME(DESTN)</td>
</tr>
<tr>
<td></td>
<td>FORMAT (14, H3.2)</td>
</tr>
<tr>
<td></td>
<td>FILE DESTN IN ROUT(ORDER)</td>
</tr>
<tr>
<td>10</td>
<td>LOOP</td>
</tr>
<tr>
<td></td>
<td>CALL ARRVL(ORDER)</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>
nature with an intentional provision of redundancy. This experimental approach also influenced the
design of the compiler, emphasizing modifiability over other factors such as reliability.

Buxton and Laski [1962, p. 198] attribute contributions to CSL from Tocher and his colleagues at
United Steel Companies, Ltd., D.E. Hartley of Cambridge University, and I.J. Cromar of IBM, who
had major responsibility in writing the compiler. They also acknowledge becoming aware of
SIMSCRIPT only on the completion of the CSL definition.

No estimates are given for the language design effort, but the compiler is estimated to have required
about nine man-months. The object program produced is described as "fairly efficient," both in terms
of storage and execution time [Buxton 1962, p. 198]. A second version of CSL, described by Buxton
[1966] and labeled C.S.L. 2 by Clementson [1966], attributed to P. Blunden, P. Grant, and G. Parncutt
of IBM UK the major credit for the compiler design that resulted in the product described by [IBM
UK 1965]. This version, developed for the IBM 7094, provided extensive dynamic checking to aid
program verification and offered some capability for parameter redefinition without forcing recom-
pilation [Buxton 1966, p. 140].

8.3.4.2 Rationale for Language Content

Influencing Factors The following factors appear to have influenced CSL:

1. the intended use for decision making beyond simulation models prompted the adoption of a
   predicate calculus approach [Buxton 1962, p. 194],
2. the reliability on FORTRAN for intermediate translation, and
3. the dependence on techniques advanced by Tocher in GSP promoting the focus on activity as
   the basic descriptive unit.

800 CHAPTER VIII
The terminology of CSL resembles that of SIMSCRIPT in that entities, attributes, and sets are primary static model descriptors. The T-cells of CSL provide the basis for dynamic description, that is, representation of time associated with the initiation and completion of activities.

**Language Design and Definition** To the FORTRAN arithmetic and matrix manipulation capability, the developers of CSL added set manipulation operations, statistical utilities, and some logical operations to support the predicate calculus requirements intended for activity tests. The term ACTIVITIES both signals the use of CSL for simulation purposes (adding of timing requirements) as well as marking the beginning of activity description.

With reference to the six language requirements for discrete event simulation, CSL provided all, but some in limited form as shown in Figure 8.6. The CSL manual contains no explicit identification of the random number generation technique although a linear congruential method is implied by the requirement of an odd integer initialization. Only the normal and negative exponential transformation techniques were supplied in addition to a user defined specification for table look-up. Class definition of entities gave a major descriptive advantage over programming in the host language (FORTRAN II or FAP). The class definition example in Figure 8.6 shows a driver class with two attributes where entities belong to one of two sets, either local or long distance (LD). The use of TIME prescribes a T-cell associated with each driver. Set manipulation is illustrated in the FIND statement which shows that a customer entity has two attributes and the search is intended to find the first in the queue that

**FIGURE 8.6**
The CSL Capabilities for Meeting the Six Requirements.

<table>
<thead>
<tr>
<th>Language Requirement</th>
<th>Basis for Provision</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Number Generator</td>
<td>Automatically Generated Function</td>
<td>TM1 = RANDOM(STREAMB,2)</td>
</tr>
<tr>
<td>Process Transformers</td>
<td>Rectangular: variable = RANDOM(stream,range)</td>
<td>SERV.T = DEVIATE(STRMC,3,30)</td>
</tr>
<tr>
<td></td>
<td>(Type of generator not specified)</td>
<td>X = NEGEXP(BASE,MEAN(2))</td>
</tr>
<tr>
<td></td>
<td>Automatically Generated Functions</td>
<td>T.MACH.K = SAMPLE(1, POISSON, STRM)</td>
</tr>
<tr>
<td></td>
<td>Normal: variable = DEVATE(stream, stddev, mean)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Negative Exponential: variable = NEGEXP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(stream, mean)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>User Defined: variable = SAMPLE(cell name, dist, name, stream)</td>
<td></td>
</tr>
<tr>
<td>List Processing Capability</td>
<td>Class definition of objects having the same attributes</td>
<td>CLASS TIME DRIVER.10(2) SET</td>
</tr>
<tr>
<td></td>
<td>that can be referenced as a set or sets with the use of</td>
<td>LOCAL,LD</td>
</tr>
<tr>
<td></td>
<td>TIME assigning a clock (T.cell) to each entity.</td>
<td>FIND CUST QUEUE FIRST &amp;120</td>
</tr>
<tr>
<td></td>
<td>Additions to (GAINS) and deletions from (LOSES) set, and searching based on FIRST,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LAST, MAX (expression), MIN (expression), ANY</td>
<td></td>
</tr>
<tr>
<td>Statistical</td>
<td>Provides only a HIST statement for accumulating data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>during program execution. FORTRAN II or FAP subroutines can be called</td>
<td></td>
</tr>
<tr>
<td>Report Generation</td>
<td>FORTRAN input/output and formatting</td>
<td></td>
</tr>
<tr>
<td>Timing Executive</td>
<td>Conditions specified which are tested. Time cells provide the link between time and</td>
<td>WAIT.CUST(1) GE 10</td>
</tr>
<tr>
<td></td>
<td>state. Based primarily on GSP [Tocher 1960]</td>
<td>WAIT.CUST(2) GE 35</td>
</tr>
</tbody>
</table>

DISCRETE EVENT SIMULATION LANGUAGE SESSION 381
Initial statements in the program read in a two-dimensional data array of mileages between all airports in a given part of the world, and establish three sets that hold subgroups of airports. The first of these, AIRPORTS, holds the names of those airports between which possible transfer routings involving one change of aeroplane are required. The second set, TRANSFERPORTS, holds the names of the major airports where transfer facilities are possible. The third set, USED, is used during the program as a working-space set. A transfer routing is permissible provided that the total mileage flown does not exceed the direct mileage by more than 15%. The following program establishes valid transfer routings. The initial statements are omitted and the output statements are stylized to avoid the introduction of detail which has not been fully described in the paper. The transfer airports for each airport pair are written out in the order of increasing total mileage.

```fortran
FOR A AIRPORTS
  FOR B AIRPORTS
    A LT B & 2
    WRITE A, B
    ZERO USED
    1 FIND X TRANSFERPORTS MIN
       (MILEAGE (A, X)+MILEAGE (X, B))
    & 2
    X NE A
    X NE B
    100* (MILEAGE (A, X)+MILEAGE (X, B))
    LE 115*MILEAGE (A, B)
    AIRPORT.X NOT IN USED
    WRITE X
    AIRPORT.X HEAD USED
    GO TO 1
  2 DUMMY
EXIT
```

has attribute 1 value greater than or equal to 10 and attribute 2 value greater than or equal to 35. Basically, conditions were tested to determine if an activity could initiate; the termination was usually prescribed through the assignment of T-cell value. The timing executive was very similar to that used by Tocher in GSP [Buxton 1962, p. 197].

Activities had a test and an action section. The test could be prescribed in rather complex logical statements comprising a chain. An evaluation of the chain would lead to the proper actions within the activity specification.

Translation was accomplished in four phases. An intermediate phase produced a FORTRAN II program as output on an IBM 1401 (serving as input tape creator for an IBM 7090). Despite the multiphase translation with repetitive passes, the authors felt the process to be efficient, easy to modify, and easy to extend [Buxton 1962, p. 198]. A ratio of CSL to FORTRAN statements is "of the order of one to five." The authors also claim that an equivalent CSL program can be written in approximately 20 percent of the time required for a FORTRAN program.

The problem statement and example of CSL code shown in Figure 8.7 are taken from Buxton [1962, p. 199].

Nontechnical Influences  CSL continued the GSP organization of a model through activities. It also continued the use of activity scan but provided a simpler two-phase as opposed to a three-phase
PAPER: A HISTORY OF DISCRETE EVENT SIMULATION PROGRAMMING LANGUAGES

executive, probably because the application domain stressed contingent (state conditioned) events more than determined (time dependent) events. Following both precedents kept CSL in the mainstream of British simulation efforts, and it soon became a favorite for program generation techniques based on the entity-cycle or activity cycle diagrams, the successors to the wheel charts. Examples of these early graphical modeling aids are shown in Figure 8.8.

8.3.5 GASP

GASP (General Activity Simulation Program) was developed by Philip J. Kiviat at the Applied Research Laboratory of the United States Steel Corporation. Kiviat came to U.S. Steel in June of 1961 and began work on the steel mill simulation project shortly thereafter [Kiviat 1991a].

8.3.5.1 Background

The initial draft of GASP and a large part of the coding occurred during 1961, originally in ALGOL. Early on (presumably in 1961) the decision was made to base the GASP simulator on FORTRAN II. By February 1962, the design of the language was completed sufficiently so that Kiviat received approval for a seminar presentation entitled, “A Simulation Approach to the Analysis of a Complex Large Scale Industrial System,” in the Department of Industrial and Engineering Administration at Cornell University. The invitation was extended by Professors Richard W. Conway and William L. Maxwell, both significant contributors to simulation conceptual development.

Developed for internal use within United States Steel, GASP was intended to run on several machines, for example, the IBM 1410, IBM 1620, Control Data 1604, and Bendix G-20. A preliminary version of the Programmer’s Manual, dated January 16, 1963, also showed intended execution on the IBM 7070 and 7090 with erasures of pencil inclusions of the IBM 7074 and 7094. Editorial markings by Kiviat noted that an update of the target machines would be necessary at final typing [Kiviat 1963a].

A report during the same time period [Kiviat 1963c] clearly casts GASP as the key ingredient in a method for utilizing computer simulation in the analysis of steel industry operations. The intended report was to educate potential users as to the benefits derived from simulation studies and to serve as an introduction to the use of the technique with a GASP “simulation-programming system that greatly reduces the time required for problem analysis, computer programming, and computer use” [Kiviat 1963c, Abstract].

8.3.5.2 Rationale for Language Content

Exposed to the subject in his graduate study at Cornell, Kiviat was both knowledgeable in simulation techniques and aware of other language developments. The completed version of the report describing GASP [Kiviat 1963b] (a revision of the earlier titled “Programmers’ Manual”) contained references to SIMSCRIPT, Tocher’s GSP, SIMPAC, and industrial dynamics [Forrester 1961]. An incomplete reference contains the name “Gordon, Geoffery [sic],” but nothing more.

Influencing Factors GASP was designed to “bridge the gap” between two groups: operating or engineering personnel, unfamiliar with computer programming, and computer programmers unknowledgeable of the application domain. Like GPSS, flow-chart symbols were intended to be used by the operational personnel in defining the system. The symbols followed the conventions for general purpose use. An example, taken from [Kiviat 1963a], is shown in Figure 8.9. The sketch of language capabilities, provided in Figure 8.10, is supplemented by the following explanation.
A crane is shared by three machines (C2, C3, C4), each requiring it to unload parts. A state of C4 can also cause a state to occur for C2.

(a) The Wheel Chart Example (from [Tocher 1966, p. 129])

A machine requires a setup operator to prepare it for production (Running). A 3-minute setup time is followed by an idle period for the operator until the next setup. The machine runs for 10 minutes and stops to await setup for the next run.

(b) The Activity Cycle Example (from [ECSL-CAPS Reference Manual 1978, p. 9])
A GASP model was made up of elements—people, machines, orders—with elements described in terms of attributes. A list of attributes describing an element corresponded to the later concept of a record. Changes in the value of attributes occur through activities, which could either be instantaneous or require a time duration. Event routines were described by the user or the operating personnel, through flow charts subsequently translated into executable representations by programmers.

Subroutines written in FORTRAN II provided the list processing capabilities (queue insertion, item removal, etc.) relying on the array data structure completely. Reporting was centralized in a single subroutine (OUTPUT) to which transfers were made, depending on the data being collected.

The FORTRAN II resident library provided the random number generator, and GASP utilized transformations for the generation of variates following the uniform, Poisson, normal, and Erlang distributions. In addition, sampling could produce a value from an empirical distribution (based on sample data).

A rather unique feature of GASP was the provision of a regression equation from which sampling could be used primarily to produce input values. The number of arguments for the regression equation was limited to 10 or less, and the error term was expected to follow a normal distribution.

Timing was the responsibility of the GASP EXECUTIVE, which utilized the data supplied by the modeler’s use of scheduling (SCHED) calls to sequence events properly. Debugging was also a major
concern, and features were provided in a special subroutine (MONITR) that assisted in this effort. A routine for error determination reported 13 types of errors.

**Nontechnical Influences** A revision to GASP, labeled GASP II, was underway in early 1963. No doubt, further developments were hindered by the departure of Kiviat who took a position at the RAND Corporation. A later version of the GASP User’s Manual, which is believed to be in the 1965 time frame, was authored by Jack Belkin and M.R. Rao [Belkin 1965?].

### 8.3.6 OPS-3

OPS-3 is classed as a major SPL, not because of conceptual influences but because it was an innovative and technical effort considerably ahead of its time. Growing out of classroom instruction at MIT during the spring, 1964, OPS-3 was a prototyping effort building on two earlier versions, each the result of classroom experiments. The MIT time sharing system, CTSS, stimulated the experiments in interactive model building and simulation programming.

#### 8.3.6.1 Background

The major contributors to OPS-3 were Martin Greenberger, Malcolm M. Jones, James H. Morris, Jr., and David N. Ness. However, a number of others are cited in the preface of Greenberger [1965]: M. Wantman, G.A. Gorry, S. Whitelaw, and J. Miller. The implication is that all in this second group were students in the Alfred P. Sloan School of Management.

The OPS-3 system was intended to be a multi-purpose, open-ended, modular, compatible support for creative researchers. The users were not intended to be computing experts, but persons involved in problem solving and research that necessitated model building. Additions to OPS-3 could be from a variety of programming languages or represent a synthesis of subprograms without modification [Greenberger 1965, pp. 1–2].

#### 8.3.6.2 Rationale for Language Content

OPS-3 was intended to be a general programming tool with extended capabilities for handling modeling and simulation to support research users. As pointed out before, OPS-3 was viewed as a system and not simply a language.

**Influencing Factors** The major influence on OPS-3 was the interactive, time-sharing environment provided by CTSS. For the first time simulation modeling was to be done in a quasi-real-time mode, but not necessarily with interactive assistance beyond that provided by CTSS. The AGENDA, provided as a compound operation (KOP), enabled the scheduling, cancellation, and rescheduling activities following the "spirit of simulation languages such as SOL and SIMSCRIPT," [Greenberger 1965, p. 7]. The power of interactive model execution was recognized as something totally new.

**Language Design and Definition** OPS-3 is structured in a hierarchical fashion with a basic calculation capability furnishing the lowest level of support. The system provides five modes or user states: Execute, Store, Store and Execute, Run, and Guide. Simple operators such as PRINT are available in execute mode. Cascading a series of operations into a program is enabled with store mode. If single operators are to execute within a program then store and execute mode was used. Run mode enabled execution of compound operations (KOPs) without interruption. The assistance or help was accessible in guide mode. In the terminology typical of the time, OPS-3 would be categorized as emphasizing...
FIGURE 8.10
The GASP Capabilities for Meeting the Six Requirements.

<table>
<thead>
<tr>
<th>Language Requirement</th>
<th>Basis for Provision</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Number Generation</td>
<td>FORTRAN II resident library, NRANDM(d) with d = 1, 2, 3, or 4 gives an integer $[1,10^d]$</td>
<td>JDGT = NRANDM(I) - 1</td>
</tr>
<tr>
<td>Process Transformers</td>
<td>Uniform, Poisson, Normal, and Erlang distributions. An empirical distribution also could be defined.</td>
<td>NUM = NPOISN (3) (where 3 specifies the row of the input parameter list which contains the actual parameter value)</td>
</tr>
<tr>
<td>List Processing Capability</td>
<td>Subroutines provided the capability for filing (FILEM) and retrieving (FETCHM) elements using FIFO or LIFO disciplines in a FORTRAN array.</td>
<td>CALL FILEM (TMDUE,JDOLR,KMATL,QUEUE,1) CALL FETCHM (FLOATF(JDOLR),XINTF(TMDUE),KMATL,QUEUE,1) (Note: first argument is the basis for insertion and removal)</td>
</tr>
<tr>
<td>Statistical</td>
<td>Subroutines COLECT: the sum, sum of squares, extreme values, and sample size for up to 20 designated variables HISTOG: forms a histogram for each up to 20 designated variables</td>
<td></td>
</tr>
<tr>
<td>Report Generation</td>
<td>The OUTPUT subroutine must be written by the user but the statistical subroutines above and PRINTQ provide the functionality.</td>
<td></td>
</tr>
<tr>
<td>Timing Executive</td>
<td>Event scheduling with a scheduling subroutine and using an associated list organization developed by Kiviat [1962a, 1962b]</td>
<td>CALL SCHDL(ARVTM,JARVL,KCUST)</td>
</tr>
</tbody>
</table>

operators rather than data structures. An operator took the form of a closed subroutine written in MAD (Michigan Algorithmic Decoder), FORTRAN, FAP (an assembler), or any other source language available under CTSS. Over 90 operators were provided in the language, and 75 subroutines were provided in the library.

In terms of the six language requirements for simulation, Figure 8.11 shows the capabilities provided by OPS-3. Especially impressive were the statistical capabilities for input data modeling and output analysis through an operator set that provided linear least-squares fit, intercorrelation and partial correlation, tests of homogeneity, contingency table analysis, and others. A guided form of each test ushered the novice user through the requirements for performing the test. A short form was available for the experienced user.

The determination of the next activity is based on a three-phase method resembling that used in Tocher’s GSP [Tocher 1960]. The contingent event is placed on the AGENDA in an order dictated by TIME, the system clock, in the sequence of conditional events. Parameters can be passed with the SCHED operator, and the AGENDA can be accessed by the user as a debugging aid.

**Nontechnical Influences** This period in MIT computing was certainly one of the most exciting. Time sharing and interactive computation were in their infancies. Innovative thinkers such as J.C.R. Licklider, John McCarthy, and R.M. Fano were pushing the boundaries of computing technology to
The OPS-3 Capabilities for Meeting the Six Requirements.

<table>
<thead>
<tr>
<th>Language Requirement</th>
<th>Basis for Provision</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Number Generation</td>
<td>Defined function, presumably a linear congruential method.</td>
<td>DRAW PRIORITY RANDOM 1 7</td>
</tr>
<tr>
<td>Process Transformers</td>
<td>The second argument of DRAW specified the distribution: exponential, normal, or modeler specified.</td>
<td>DRAW SERVTM EXPONE THETA</td>
</tr>
<tr>
<td>List Processing Capability</td>
<td>Major control through user-accessible symbol table.</td>
<td>RESETQ WLINE</td>
</tr>
<tr>
<td>Statistical Report Generation</td>
<td>Extensive statistical operators with capability for input data modeling and output analysis</td>
<td>TTESTS X N1 N2</td>
</tr>
<tr>
<td>Timing Executive</td>
<td>A three-phase method built around the KOP AGENDA.</td>
<td>SCHED SRVCE AT 100</td>
</tr>
</tbody>
</table>

In their static form, computer-program models are documents. They preserve and carry information just as documents printed in natural language do, and they can be read and understood by recipients who know the modeling language. In their dynamic form, however, computer-program models appeal to the recipient’s understanding directly through his perception of dynamic behavior. That model of appeal is beyond the reach of ordinary documents. When we have learned how to take good advantage of it, it may—indeed, I believe it will—be the greatest boon to scientific and technical communication, and to the teaching and learning of science and technology, since the invention of writing on a flat surface.

Within less than two years of publication, the DEC PDP-10 would supply even more advanced facilities than afforded to the developers of OPS-3. The efforts of Greenberger, Jones, and others with OPS-3 deserve recognition for the perceptions and demonstration of technology potential rather than the effect on SPL developments.
8.3.7 DYNAMO

The DYNAMO (DYNAmic MOdels) language served as the executable representation for the industrial dynamics systems models developed by Jay Wright Forrester and others at MIT during the late 1950s and into the 1960s and beyond. DYNAMO is the lone nondiscrete event simulation language included in this history. Justification for this departure is that concept, techniques, and approaches utilized by the MIT group had a significant impact on those working in the development of discrete event SPLs [Kiviat 1991b].

8.3.7.1 Background

The predecessor of DYNAMO, a program called SIMPLE (Simulation of Industrial Management Problems with Lots of Equations), was developed by Richard K. Bennett for the IBM 704 computer in the spring of 1958. (Note that this is a contemporary of GSP in the discrete-event SPL domain.) SIMPLE possessed most of the basic features of DYNAMO, but the model specifications from which the program was derived were considered to be rather primitive [Pugh 1963, p.2]. DYNAMO is attributed to Dr. Phyllis Fox and Alexander L. Pugh, III with assistance from Grace Duren and David J. Howard. Modifications and improvement of the original SIMPLE graphical plots was provided by Edward B. Roberts [Forrester 1961, p.369].

Originally written for an IBM 704 computer, the DYNAMO compiler, consisting of about 10,000 instructions, was converted to an IBM 709 and then an IBM 7090 by Pugh. The DYNAMO compiler as described in the User's Manual represented about six staff-years of effort (including that required to develop SIMPLE), and the maximum model size was about 1500 equations. A FORTRAN simulator (FORDYN), intended for users who did not have access to the large IBM 7090 or 7094, was developed by Robert W. Llewellyn of North Carolina State University in 1965 [Llewellyn 1965]. (Note that neither DYNAMO nor FORDYN are included in Figure 8.1.)

8.3.7.2 Rationale for Language Content

Although applicable to the modeling of any information feedback system, DYNAMO was developed as the means of implementing industrial dynamics models, which addressed the application of simulation to large scale economic and social systems. The level of modeling granularity does not address the individual items or events typically associated with the job shop models of that time. A thorough understanding of the basis for systems dynamics modeling, the term later employed, is beyond the scope of this paper (see [Forrester 1961]).

Influencing Factors The design of DYNAMO was strongly influenced by the following desirable properties [Pugh 1963, p.2]:

1. The language should be easily understood and easy to learn by the user group, who in general might not be professional programmers. Flexibility would be sacrificed for both ease of use and correctness.

2. A very efficient (at the time) compilation phase, with no object language production, eliminated the usual practice at the time of saving object text.

3. Output in the form of graphical plots was recognized as more useful than extensive numerical results; however, both were provided to the user.
4. Error detection and error correction were considered a primary responsibility. Both initial conditions and the order of computation were subject to detection and correction. The claim was that DYNAMO checked for almost every logical inconsistency, and provided comments on errors that were easily understood.

**Language Design and Definition** DYNAMO is described as a language expressing zero- and first-order difference equations. A system is described in terms of flows and accumulations. Flows in orders, material, personnel, money, capital equipment, and most importantly information, affect the decision making that leads to changes in the rate of flow occurring in subsequent time periods. Over time the changes in flow contribute to the redefinition of levels (of money, materials, information) that have subsequent influence on future rates of flow. Each model must begin with an initial state definition, and the constant increment of time (DT) is selected to be sufficiently small to avoid inherent instability.

Figure 8.12 portrays the timing sequence for each set of DYNAMO computations. The present time (K), has values determined based on levels of accumulation at J modified by rates of flow over the JK interval. Levels at present time (K) then contribute to the definition of rates over the interval KL, where L = K + DT.

Figure 8.13 is an example taken from [Pugh 1963, p.17] showing the model of a retail store that illustrates the DYNAMO representation. The equational representation in Figure 8.13 is understood by noting that the number of each equation is suffixed by the type of equation: L for level; A for auxiliary; R for rate; N for initial value. NOTE is a comment statement, and special statements such as PRINT, PLOT, and SPEC provide instructions on the execution process. The subscripting enables not only a determination of direct statement formulation, but also of statement ordering and typing. Although discontinuities were permitted in systems dynamics, and could be represented in DYNAMO, they were discouraged unless absolutely necessary as a modeling technique.

The graphical output produced on a lineprinter consisted of letters showing the relative values for designated variables. The use of plotters with smoothed curvilinear, coded representations is provided in the book by Forrester, but was not available with the early version of DYNAMO. The fixed format field definition required by the translator was to be replaced in later versions.


**PAPER: A HISTORY OF DISCRETE EVENT SIMULATION PROGRAMMING LANGUAGES**

**FIGURE 8.13**

Listing of model.

```
* M478-248,DYN,TEST,1,1,0,0
RUN 2698JP
NOTE MODEL OF RETAIL STORE
NOTE
1L IAR.K=IAR.J+(DT) (SRR.JK-SSR.JK) INVENTORY ACTUAL
1L UOR.K=UOR.J+(DT) (RRR.JK-SSR.JK) UNFILLED ORDERS
20A NIR.K=IAR.K/DT NEGATIVE INVENTORY
20A STR.K=UOR.K/DFR SHIPMENTS TRIED
54R SSR.KL=MIN(STR.K,NIR.K) SHIPMENTS SENT
40R PSR.KL=RRR.JK+(1/DIR) (IDR.K-IAR.K) PURCHASE ORDERS SENT
12A IDR.K=(AIR) (RSR.K) INVENTORY DESIRED
3L RSR.K=RRR.J+(DT) (1/DRR) (RRR.JK-RSR.J) REQUISITIONS SMOOTHED
39R SSR.KL=DELAY3(PSR.JK,DTR) SHIPMENTS RECEIVED
NOTE
NOTE INITIAL CONDITIONS
NOTE
12N UOR=(DFR) (RRR)
6N RSR=RRR
6N IAR=IDR
NOTE INPUT
NOTE
7R RRR.KL=RRR+RCR.K REQUISITIONS RECEIVED
45A RCR.K=STEP(STH,5) REQUISITION CHANGE
NOTE
NOTE CONSTANTS
NOTE
C AIR=8 WKS CONSTANT FOR INVENTORY
C DFR=1 WK DELAY IN FILLING ORDERS
C DIR=4 WKS DLY REFILLING INVENTORY
C DRR=8 WKS REQUISITION SMTHNG T C
C DTR=2 WKs DELAY IN TRANSIT
C RRI=1000 ITEMS/WK Req. RECEIVED INITIALLY
C STH=100 ITEMS/WK STEP HEIGHT
NOTE PRINT 1)IAR,DIR/2)UOR/3)RRR,SSR/4)PSR,SRR
PLOT IAR=I,UOR=U/RRR=R,SSR=S,PSR=P,SRR=Q
SPEC DT=0.1/LENGTH=10/PRTPER=5/PLTPER=0
```

**Nontechnical Influences**  Systems dynamics and DYNAMO, although developed totally at MIT, attracted considerable nationwide interest. Financial support from the Ford Foundation and the Sloan Research Fund are acknowledged in the Preface of Forrester [1961]. Additional support from a number of sources is identified there also.

Since DYNAMO was not a discrete event SPL, and because its influence was limited to the developmental period, successive versions of the language are not described. However, the language continued, both in refined versions [Pugh 1973] and extensions [Pugh 1976].

**DISCRETE EVENT SIMULATION LANGUAGE SESSION** 391
8.3.8 Other Languages

Numerous SPLs emerged during this advent period. Most, experienced some use and disappeared. A few had distinctive characteristics that deserve some note in a paper of this type.

MILITRAN was produced by the Systems Research Group for the Office of Naval Research [Systems Research Group, Inc. 1964]. Krasnow [1967, p. 87] states that little concession to military subject matter was given in MILITRAN. Event scheduling was the world view promoted, and a distinction between permanent (synchronous) and contingent events was made.

The Cornell List Processor (CLP) was a list processing language used extensively for simulation instruction. The developers had the goal of producing a list processing language that required no more than a "FORTRAN level" knowledge of programming [Conway 1965]. CLP relying on CORC, a general algebraic language used at Cornell also for instructional purposes, enabled students to define entities, and use them in set manipulation (INSERT, REMOVE) without the major effort of learning a language such as SIMSCRIPT [Conway 1965, p. 216]. The student still had to write his or her own timing, statistical analysis, and report generation routines.

QUIKSCRIPT was a SIMSCRIPT derivative simulator, a set of subroutines, based on the 20-GATE algebraic language used at Carnegie Institute of Technology [Tonge 1965]. The GATE subroutines did not provide all the facilities of SIMSCRIPT (e.g., no report generation), and the definitional forms of the latter were omitted.

SIMPAC, produced at the System Development Corporation, was distinguished by its fixed-time-increment timing routine. This would appear to make it the only US language to adhere completely to the activity scan world view [Bennett 1962]. (OPS-4 departed from OPS-3 by accommodating all three world views.)

SOL (Simulation Oriented Language) was developed by Knuth and McNeley as an extension to ALGOL. The language is structured much like GPSS, even using terms such as SEIZE, RELEASE, and ENTER. Sets are represented as subscripted variables. In contrast with SIMULA, SOL focused on the dynamic interaction of temporary objects, again much like GPSS [Knuth 1964a, 1964b]. Of note in SOL was the explicit use of a wait on state condition that was not present in GPSS or in SIMULA, because the prevailing view was that such an indeterminate expression could lead to gross inefficiencies (see [Nygaard 1981, p. 452] for a discussion specific to SIMULA and [Nance 1981] for a more general discussion). A second, more anecdotal, item related to SOL was the response of Knuth when asked at the IFIP Conference what were the plans for SOL; he replied that there were no plans for he found SIMULA to have all the capabilities of SOL (except for the wait on state condition) and more [Knuth 1992].

8.3.9 Language Comparisons

The end of the advent period marked the beginning of a period of intense interest in simulation programming language comparison and evaluation. Such interest was manifested in the publication of a number of papers during the 1964-1967 timeframe that reviewed and compared existing languages with respect to many criteria [Young 1963; Krasnow 1964; Tocher 1965; Kiviat 1966; Teichrow 1966; Krasnow 1967; Reitman 1967]. These sources also refer to lesser known languages that are not mentioned here.

Interest in comparison and evaluation was likely stimulated by the perceived cost associated with using "another language." The cost of acquiring the translator was minor; the cost of training people, maintaining the language, and supporting its migration to subsequent hardware systems could be very high. For that reason, packages in a general purpose language had considerable appeal.
This sharp interest in language comparisons is also marked by the number of workshops, symposia, and conferences addressing SPL issues. Daniel Teichrow and John F. Lubin are cited by Philip Kiviat as the “honest brokers” in that they had no commitment to a particular language but were instrumental in developing forums for the exchange of concepts, ideas, and plans. A workshop organized by them and held at the University of Pennsylvania on March 17–18, 1966 refers to an earlier one at Stanford in 1964 [Chrisman 1966]. Session chairs at the 1966 workshop included Kiviat, R. L. Sisson (University of Pennsylvania), Julian Reitman (United Aircraft), and J.F. Lubin. Comparison papers were presented by Harold G. Hixson (Air Force Logistics Command), Bernard Backhart (General Services Administration), and George Heidorn (Yale University). Among the 108 attendees were several who were involved in SPL development: W.L. Maxwell of Cornell (CLP), Malcolm M. Jones of MIT (OPS), Howard S. Krasnow and Robert J. Parente of IBM, and Julian Reitman of Norden (GPSS/Norden).

In between the two workshops described was the IBM Scientific Computing Symposium on Simulation Models and Gaming held at the T.J. Watson Research Center in Yorktown Heights, New York on December 7–9, 1964. A session entitled, “Simulation Techniques” included a paper by Geoffrey Gordon that compares GPSS and SIMSCRIPT [Gordon 1966] and a paper by Tocher that presents the wheel chart as a conceptual aid to modeling [Tocher 1966]. In another session, the Programming by Questionnaire (PBQ) technique for model specification is described [Geisler 1966]. (More about PBQ is to follow.) The Symposium took a broad applications view of simulation, and among the 175 attendees at this conference were Jay W. Forrester (MIT), who described the principles of industrial dynamics, Richard M. Cyert (CIT), Herbert A. Simon (CIT), Philip Morse (MIT), J.C.R. Licklider (MIT), Harold Guetzkow (Northwestern), Richard W. Conway and William L. Maxwell (Cornell), Oscar Morgenstern (Princeton), and Guy H. Orcutt (Wisconsin).

The most notable technical conference on simulation languages was the IFIP Working Conference on Simulation Programming Languages, chaired by Ole-Johan Dahl and held in Oslo, 22–26 May 1967. The Proceedings were edited by John N. Buxton and appeared the following year. The list of presenters and attendees reads like a “Who’s Who,” not only in computer simulation but also in computer science. The “by invitation only” participants, with their role or presentation subject in parentheses, included Martin Greenberger and Malcolm M. Jones (OPS-4), Michael R. Lackner (graphic forms for conversational modeling), Howard S. Krasnow (process view), Donald E. Knuth (Session Chair), Jan V. Garwick (Do we need all these languages?), R.D. Parslow (AS: an Algol language), Ole-Johan Dahl and Kristen Nygaard (class and subclass declaration), L. Patrone (SPL: a simulation language based on PL/I), John G. Laski (interactive process description and modeling languages), G.K. Hutchison (multiprocessor system modeling), Robert J. Parente (simulation-oriented memory allocation), G. Molner (self-optimizing simulation), G.P. Blunden (implicit interaction), John L. McNeley (compound declarations), C.A.R. Hoare (Session Chair), A.L. Pugh III (DYNAMO II), T.B. Steel, Jr. (standardization), and Evzen Kindler (COSMO). The highly interactive group of participants included: Richard W. Conway, Douglas T. Ross, Christopher Strachey, Robin Hills, and John N. Buxton. Issues and problems surfacing in this symposium on occasion resurface, for example, in the annual Winter Simulation Conferences.

In November 1967, Harold Hixson, Arnold Ockene, and Julian Reitman collaborated to produce the Conference on Applications of Simulation Using GPSS held in New York. Hixson, representing SHARE (the IBM User Group), served as General Chair, Ockene of IBM served as Publicity Chair, and Reitman of Norden Systems was the Program Chair. In the following years this conference expanded its programming language scope (beyond GPSS) and became known as the Winter Simulation Conference (WSC), which celebrated its twenty-fifth anniversary in December 1992, in Washington. The three individuals named above, together with other pioneers in the early years of the
Following the bustle of activity surrounding the emergence of new SPLs, the period from 1966–1970 marked a consolidation in conceptual clarification. The concepts, possibly subjugated earlier in the necessities of implementation, were reviewed and refined to promote more consistent representation of a world view and improve clarity in its presentation to users. Nevertheless, rapid hardware advancements and vendor marketing activities forced some languages, notably GPSS, to undergo major revisions.

8.4.1 GPSS

GPSS/360 extended the prior version (GPSS III) by increasing the number of block types from 36 to 44. Set operations were expanded by the introduction of groups. Extensions to the GENERATE block improved storage use. A HELP block permitting access to routines in other languages was provided. Comparisons of the language versions for GPS K, Flow Simulator, and GPSS III with GPSS/360 can be found in the Appendices of Greenberg [1972].

GPSS V provided more convenience features but made no major changes in the language. A run timer could be set by the modeler, extensions were made to the HELP block, and free format coding (removal of statement component restrictions to particular fields) was permitted. A detailed comparison of differences in GPSS/360 and GPSS V is provided in Appendix A of Schriber [1974].

8.4.2 SIMULA 67
The realization of shortcomings in SIMULA I and the influences of language and translator developments during the mid-1960s led to an extensive revision of the language. This revision, described in detail in Nygaard [1981], is not repeated here. Needless to say, the class concept was a major innovative contribution. Clearly, SIMULA 67 served to crystallize the process concept, the coroutine and quasi-parallel processing capabilities and demonstrate the implementation of abstract data types. The result was a language well beyond the power of most of its contemporaries.

8.4.3 SIMSCRIPT II
SIMSCRIPT II, although dependent on SIMSCRIPT I.5 for its basic concepts of entity, attribute, and set, was clearly a major advancement over the earlier version. SIMSCRIPT II is intended to be a
general purpose language whereas SIMSCRIPT I.5 was intended to be a simulation programming language. SIMSCRIPT II in appearance looks much different from SIMSCRIPT I [Markowitz 1979, p. 28]. An expressed goal of SIMSCRIPT II was to be a self-documenting language. SIMSCRIPT II was written in SIMSCRIPT II, just as its predecessor was written, for the most part, in SIMSCRIPT I.5.

8.4.3.1 Background

The RAND Corporation provided the organizational support and financial underpinnings for SIMSCRIPT II. Philip J. Kiviat, having come from the United States Steel Company, took over the leading role in the design and development of the language from Harry Markowitz, who had started the project. Markowitz, whose ideas had formed the basis for SIMSCRIPT I and SIMSCRIPT I.5, was in the process of leaving RAND during the major part of the language project. Bernard Hausner, the principal programmer on SIMSCRIPT I.5, had departed, and Richard Villanueva assumed this role.

Although the claim was made that SIMSCRIPT II was a general purpose language, RAND’s principal interest in developing SIMSCRIPT II was to enhance its capability within discrete event simulation and to offer greater appeal to its clients [Kiviat 1968, p. vi]. Many of RAND’s models were military and political applications, necessitating extremely large complex descriptions. The intent was to create a language that, through its free-form and natural language appearance, would encourage more interaction with application users.

Contributions to the language are acknowledged in the preface to the aforementioned book. George Benedict and Bernard Hausner were recognized as contributing much of the basic compiler design and programming. Joel Urman of IBM influenced the language design as well as programmed much of the I/O and operating system interface routines. Suggestions and criticisms were attributed to a number of persons, including Bob Balzer, John Buxton, John Laski, Howard Krasnow, John McNeley, Kristen Nyggard, and Paula Oldfather [Kiviat 1968, p. vii].

In addition to a free-form, English-like mode of communication, the language designers desired a compiler to be “forgiving,” and to correct a large percentage of user syntax errors. Furthermore, forced execution of a program was felt to reduce the number of runs to achieve a correct model implementation. Debugging statements and program control features were central to the translator.

8.4.3.2 Rationale for Language Content

Influencing Factors  Certainly, SIMSCRIPT I.5 and the entity, attribute, and set concepts had the major influence on the language design. Nevertheless, the intent to involve application users and to provide a language working within SYSTEM/360 and OS/360, both still somewhat in development, had some impact. At one point, the language designers considered writing SIMSCRIPT II in NPL (the New Programming Language), subsequently PL/I, but the idea was rejected because of the instability of NPL [Kiviat 1992]. Kiviat [1991b] acknowledges that ideas and information came from interaction with other language developers and designers. The Working Conference on Simulation and Programming Languages, cited above, was a primary source of such ideas [Buxton 1968].

Language Design and Definition  The design of SIMSCRIPT II can be represented by the reverse analogy of “peeling the onion.” The language designers refer to “levels of the language,” and use that effectively in describing the rather large language represented by SIMSCRIPT II.
Level 1 is a very simple programming language for doing numerical operations. It contains only unsubscripted variables and the simplest of control structures with only rudimentary input (READ) and output (PRINT) statements.

Level 2 adds subscripted variables, subroutines, and extended control structures to offer a language with roughly the capability of FORTRAN 66 but lacking the FORTRAN rigidities.

Level 3 adds more general logical expressions with extended control structures, and provides the capability for storage management, function computations, and statistical operations.

Level 4 introduces the entity, attribute, set concepts needed for list processing. The pointer structures, implied subscripting, and text handling go well beyond the capabilities of SIMSCRIPT 1.5.

Level 5 contains the dynamic capabilities necessary for simulation: time advance, event processing, generation of statistical variates, and output analysis.

Nontechnical Influences Markowitz [1979] attributes the lack of interest by RAND in developing SIMSCRIPT II as more than a simulation language to dictating the limitation of five levels in the implementation. He identifies Level 6 as that which dealt with database entities, and Level 7 as a language writing "language," used in the implementation of SIMSCRIPT II so that a user could define the syntax of statements and the execution of more complex commands [Markowitz 1979, p. 29]. Kiviat recalls discussion of the extension but that no concrete proposal was ever made [Kiviat 1992].

8.4.3.3 Variants

SIMSCRIPT I.5 was developed as a commercial product by Consolidated Analysis Centers, Incorporated (CACI). SIMSCRIPT II became a commercial product in SIMSCRIPT II Plus through Simulation Associates, Incorporated, cofounded by P.J. Kiviat and Arnold Ockene, formerly IBM GPSS Administrator. CACI purchased the rights to SIMSCRIPT II Plus and marketed it as SIMSCRIPT I1.5. Markowitz [1979, p. 29] also describes a SHARE version of the translator.

8.4.4 ECSL

ECSL, or E.C.S.L. as Clementson [1966] preferred, was developed for Courtaulds Ltd. by the originators of CSL [Buxton 1964]. The extensions that distinguish ECSL from its predecessor are likened to those of CSL2 developed by IBM United Kingdom, but the provision of the features took forms quite different on the Honeywell 400 and 200 series machines. A single man-year of effort was required for each of the Honeywell versions (400 and 200) [Clementson 1966, p. 215].

The most striking difference in CSL and ECSL is the departure from FORTRAN taken with the latter. This decision was based on the desire to facilitate use of ECSL by those having no knowledge of any other programming language [Clementson 1966, p. 215]. The approach taken was to adopt a columnar field formatting akin to that used in GPSS. Abandonment of FORTRAN also led to a richer I/O capability. The FORTRAN I/O was felt to be "the major shortcoming of C.S.L., as originally conceived," [Clementson 1966, p. 218]. ECSL became the target language for the Computer Aided Programming System (CAPS), the first interactive program generator, developed by Clementson in 1973 [Mathewson 1975].
A contrary argument to the separation from FORTRAN was raised almost a decade later in the simulation package EDSIM, which claimed both to emulate ECSL and to be event based [Parkin 1978]. In actuality, EDSIM closely followed the ECSL design as an activity scan language.

8.4.5 GASP II

GASP II appears twice in the genealogical tree in Figure 8.1. A preliminary description of the revised version appears in manual form available from Pritsker at Arizona State University [Pritsker 1967]. Although the published description appears in the book by Pritsker and Kiviat [1969], a listing of FORTRAN subprograms designated as a GASP II compilation on the IBM 7090 is dated “3/13/63.” This early revision, the work of Kiviat alone, predates his departure from U.S. Steel by only a few months.

An examination of the 1963 and 1967 versions of GASP II reveals both the addition and elimination of routines. The most prominent addition is the routine SET. The use of NSET as the primary data structure improved the organization of the package. A basic and extended version of GASP II are described [Pritsker 1967, pp. 22–23].

A version of GASP II, written in the JOSS interactive language, called JASP was developed by Pritsker during the summer of 1969 at RAND. Pritsker notes that memory limitations of 2000 words forced data packing and overlay techniques developed by Lou Miller of RAND. JASP is the only example of a time-sharing simulation language developed outside of MIT prior to 1970.

8.4.6 OPS-4

Defined in the Ph.D. thesis of Malcolm M. Jones [1967], the description of OPS-4 is taken here primarily from [Greenberger 1968]. Based on PL/I, OPS-4 encouraged the incremental construction and test of model components. All three world views (event scheduling, activity scan, and process interaction) were supported in the language. Additional routines for the generation of random variates were provided, and the numerous statistical routines of OPS-3 were continued. Extensive debugging and tracing capabilities were available along with immediate on-line diagnostic explanations for error detection. Model execution could be interrupted for examination and redefinition of values.

OPS-4 was described as a project in the planning stage. Intended for operation under the MULTICS operating system, OPS-4 raised the issue of the degree to which an SPL might use underlying operating systems support for simultaneous execution and asynchronous processing [Greenberger 1968, p. 22, 24]. Ambitious in scope, OPS-4 did not become a commercial language, and little is known about its eventual use.

8.4.7 Other Languages

BOSS (Burroughs Operational Systems Simulator) was initiated in 1967 as the generalization of a gun placement simulator developed by the Defense, Space, and Special Systems Group of the Burroughs Corporation. Principals in the development of BOSS were Albert J. Meyerhoff, the informal group leader, Paul F. Roth, the conceptual designer, and Philip E. Shafer, the programmer. Supported at a high level by independent research and development funds, BOSS was viewed as a potential competitor with GPSS. The first prototypes for external use were completed in early 1969. Roth [1992] acknowledges experience with GPSS II as contributing to his design of BOSS, and the
transaction flow and diagrammatic specification of GPSS are evident in the language description [Meyerhoff 1968?]. An extensive set of symbols enabled a wide variety of graphical structures to be represented, and the subsequent diagram was transformed into an executable ALGOL program. Roth [1992] attributes an internal competition with SIMULA as the reason that BOSS was never converted into a marketable product, which was the motivation of the developing group. A very interesting application of BOSS during the 1970s was to message/communication processing by CPUs in a network model developed by Scotland Yard. Roth [1992] believes the language continued in use until approximately 1982.
Q-GERT, appearing in the early 1970s, is described by Pritsker as the first network-oriented simulation language [Pritsker 1990, p. 246]. It is based on the GERT network analysis technique developed by Pritsker, and is first reported in Pritsker and Burgess [1970]. The intent with Q-GERT was to provide a user friendly modeling interface with a limited number of node types that could be combined through edge definition to describe routing conditions. Obviously, Pritsker's experience with GASP II had an influence, and the GERT family tree shown in Figure 8.14 shows the mutual influence between GERT and GASP.

Buxton [1968] contains descriptions of several languages offered in concept or perhaps developed to some extent during this period. Presentations at the 1967 IFIP workshop included: SPL (Petrone), AS (Parlow), SLANG (Kalinichenko).

8.5 THE EXPANSION PERIOD (1971–1978)

The title for this period in SPL development emanates from the major expansions and extensions to GPSS, SIMSCRIPT II.5, and GASP. The program generator concept, initiated in the U.S., took hold in the U.K. and became a major model development aid.

8.5.1 GPSS

The major advances in GPSS came from outside IBM as the success of the language caused others to extend either its capabilities or the host machine environment.

8.5.1.1 GPSS/NORDEN and NGPSS

Advances in GPSS were underway by the Norden Division of United Aircraft Corporation in the late 1960s. Labeled GPSS/NORDEN, this version of the language is an interactive, visual, on-line environment for executing GPSS models. Programmed in COBOL (the language known best by the programming team) [Reitman 1977?], GPSS/NORDEN permitted the user to interact through a CRT terminal, examining the behavior of the model (queues forming at facilities, storages filling and emptying, etc.) during execution. The user could interrupt model execution and redefine certain standard numerical attributes and then resume execution. The NORDEN report generator was described as a radically new free format English-like compiler that provides an ideal method for intermixing data, text, and titles [GPSS/NORDEN 1971, pp. 1-3].

GPSS/NORDEN in effect was a redesign around the matrix organization of GPSS. Model redefinition was accomplished through manipulation of matrices, and interactive statistical display and manipulation routines enabled both the inspection and alteration of matrix data, supported by the VP/CSS time sharing system. A film showing the use of this version of the language was distributed by the United Aircraft Corporation during the 1970s.

A version of the language entitled NGPSS (Norden GPSS) is described in a report dated 15 December 1971. Characterized as a superset of GPSS/360 and GPSS V for both batch and interactive versions, the User's Guide attributes the translator with the capability of handling very large databases, utilizing swapping capability to IBM 2311 or 2314 disk drives. A limited database capability is provided through the GPSS language to permit a modeler's access to matrix savevalues so that model components can be created and stored in a library [NGPSS 1971]. The User's Guide is replete with CRT displays of program block structures, data input, and model execution. A light pen is illustrated as used with the display [NGPSS 1971, p. 71].
8.5.1.2 GPSS V/6000

A version of GPSS V was developed by Northwestern University for Control Data Corporation in a project completed in 1975. Program copyright is shown in [GPSS V/6000 1975]. GPSS V/6000 was intended to be fully compatible with the IBM product bearing the name GPSS V. In some cases perceived restrictions imposed in the IBM version were removed in order to allow compatibility with an earlier version GPSS III/6000 [GPSS V/6000 1975, p. v-1]. The translator resided under the SCOPE 3.3 operating system (or later versions) and was a one-pass assembler in contrast to the two-pass IBM version. Differences between the two versions are described in detail in Appendix B of [GPSS V/6000 1975].

8.5.1.3 GPDS and GPSS 1100

A version of GPSS, based apparently on GPSS/360, called the General Purpose Discrete Simulator, was a program product of Xerox Data Systems for the Sigma Computer line. Little is known about it beyond the identification in [Reifer 1973]. Similarly, a UNIVAC product labeled GPSS 1100 is referenced with a 1971 date [UNIVAC 1971a, 1971b]. Little else is known about this version, particularly, if it was related in any way to GPSS/UCC, which was also implemented on a UNIVAC 1108.

8.5.1.4 GPSS/H

In 1975, James O. Henriksen published a description of a compiler for GPSS that produced a 3:1 performance enhancement [Henriksen 1976]. With the announcement of the termination of support for GPSS V by IBM in the mid-1970s, Henriksen was poised to market his version of the language which was already gaining proponents. In 1977 GPSS/H was released by Wolverine Software Corporation, which have supported the language with extensive enhancements since that time [Schriber 1991, p. 17]. Other versions of the language have been developed, both on PCs and mainframes, but GPSS/H (the “H” for Henriksen) remains to date the most popular version.

8.5.2 SIMULA 67

The historical description in [Wexelblatt 1981] describes the period of expansion for SIMULA as the development of a pure system description language called DELTA [Holbaek-Hanssen 1977]. Beginning from the SIMULA platform, the DELTA Project sought to implement system specification for simulation execution through a series of transformations from the high-level user perspective, represented in the DELTA specification language, through an intermediate language called BETA, culminating with an executable language called GAMMA. Probably due to lack of funding, the project is not viewed as having been successful.

Beyond the recognition of the lack of success of the DELTA project, little can be said conceptually about SIMULA during this period. However, the originating organization (Norwegian Computing Center) issued publications that noted comparisons of the language with other SPLs and sought to promote the language in this way [Virjo 1972; Røgeberg 1973]. Houle and Franta [1975, pp. 39–45] published an article showing that the structural concepts of SIMULA encompassed those of other languages, notably GPSS and SIMSCRIPT, and included a comparison with the language ASPOL [MacDougall 1973]. They specifically claim that GPSS and SIMSCRIPT concepts are a subset of the structural capabilities of SIMULA.
8.5.3 SIMSCRIPT

As the decade of the seventies opened, simulation languages provided contrasting positions in the representation of world views identified by Kiviat in the classic RAND report [1969]:

- event scheduling—SIMSCRIPT II, GASP II,
- activity scan—ECSL, and
- process interaction—GPSS V, SIMULA 67.

However, those studying model and language representation concepts readily recognized a major distinction between GPSS and SIMULA in the representation of the process interaction world view, that is, GPSS lacked the flexibility of SIMULA in portraying interactions among permanent entities, which made the former cumbersome for problems where such relationships must be represented.

The expansion period marked the beginnings of the breakdown in conceptual distinctions among languages. Nowhere was this more apparent than with SIMSCRIPT II.

Supported by the U.S. Department of Agriculture, Agricultural Research Service, CACI completed the extension of SIMSCRIPT II.5 to include continuous simulation components in 1976. This effort is documented with a report authored by Claude M. Delfosse [1976] describing C-SIMSCRIPT. This report describes the definitions and program structure for continuous simulation and integration control variables within SIMSCRIPT II.5. Examples of both continuous and combined models are given.

Almost concurrently, the process view with focus on temporary entities was added to the language. Using the distinction of processes to describe the typical GPSS transaction and resources to represent facilities and storages, the modeling approach enabled a definition of resources as passive (permanent) entities and temporary entities represented as processes. Process instances are created similar to the creation and scheduling of events. An ACTIVE or CREATE statement is used. Within the process a WORK or WAIT statement provides for time passage, the latter with a process in a passive, the former an active, state. Processes may SUSPEND, INTERRUPT, or RESUME other processes.

A second form of expansion took place in the form of computer system modeling languages based on SIMSCRIPT. Two such vehicles were ECSS [Kosy 1975] and CSP II [Iwata 1975]. Developed at the RAND Corporation, ECSS became a favored language for computer system performance evaluation studies.

8.5.4 GASP

During this period GASP was the subject of notable changes and considerable experimentation by Pritsker and his graduate students at Purdue. These activities are described extensively in Pritsker [1990].

8.5.4.1 GASP IV

GASP IV became available on a published basis with a book in 1974 [Pritsker 1974]. The contributions of Nicholas R. Hurst and C. Elliott Sigal are noted in the preface to the book. GASP IV proceeded philosophically in the differentiation between state and time as modeling characterizations. State events and time events became means of describing the event scheduling concept of an event and the activity scan concept of a condition. However, the concept of state variables was used to extend the discrete event capability to a continuous representation.
RICHARD E. NANCE

Mirroring the SIMSCRIPT II strategy, GASP augmented the transaction flow world view in the 1975–1976 time period [Washam 1976a, 1976b]. In [Pritsker 1990, p. 252] GASP is described as an on-going language development that is intended to produce subsequent versions at least up to GASP VI.

The increased capability and flexibility of treating both continuous and discrete models within the same language provided a claimed advantage for GASP over its contemporaries. This advantage was short-lived because SIMSCRIPT followed with the capability in 1976 [Delfosse 1976].

8.5.4.2 GASP_PL/I

In the early 1970s, several efforts were made to map a simulation language onto PL/I. Some of these are noted in the final section describing this period. GASP_PL/I, developed by Pritsker and Young [Pritsker 1975] produced a version of GASE A doctoral student examining the coded subroutines commented at the time that it appeared as if a FORTRAN programmer had done a statement by statement syntactic translation of FORTRAN into PL/I, and none of the PL/I features were utilized. Kiviat notes that preoccupation with a PL/I implementation was rather widespread at this time [Kiviat 1992].

8.5.5 Program Generators

Simulation program generators were intended to accept a model definition in a nonexecutable form and produce an executable program or one that would admit execution after slight modification. In a sense the early definitional forms of SIMSCRIPT suggested such an approach, replacing the 80-column card specification with custom forms for entity definition and report generator layout. The first program generator was Programming By Questionnaire (PBQ), developed at the RAND Corporation [Ginsberg 1965; Oldfather, 1966]. With PBQ model definition was accomplished through a user's completion of a questionnaire. The questionnaire, with the user's responses, forms a specification that produces a SIMSCRIPT program.

While PBQ was under development at RAND, K.D. Tocher was showing the utility of “wheel charts” for model description in GSP [Tocher 1966]. The “wheel chart,” based on the UK “machine based” (permanent entity) approach, pictured the cyclic activity of machines going from idle to busy as they interacted with material requiring the machine resource. In this same paper Tocher comments on the potential for real-time simulation for process control.

Interactive program generators, those permitting an interactive dialogue between modeler and program generator, appeared in the early 1970s in the UK. CAPS-ECSL (Computer Aided Programming System/Extended Control and Simulation Language), developed by Alan T. Clementson at the University of Birmingham, is generally considered the first [Mathewson 1975]. DRAFT, developed by Stephen Mathewson at Imperial College, appeared in several versions in which the activity cycle diagrams (also known as entity cycle diagrams), successors to the “wheel charts,” are translated into DRAFT/FORTRAN, DRAFT/GASP, DRAFT/SIMULON, or DRAFT/SIMULA programs [Mathewson 1977].

Two efforts, not specifically program generators but related to them, deserve mention in this section. The first is HOCUS, a very simple representational form for simulation model specification, first suggested by Hills and Poole in 1969 but subsequently marketed both in the U.S. and the U.K. in the early to mid-1970s [Hills 1969]. The second is the natural language interface to GPSS, a project of George E. Heidorn at the Naval Postgraduate School in the early 1970s. Heidorn's work actually began at Yale University in 1967 as a doctoral dissertation [Heidorn 1976]. The project had as its eventual goal to enable an analyst working at a terminal to carry on a two-way dialogue with the
computer about his simulation problem in English. The computer would then develop the model, execute the simulation, and report the results, all in an English dialect [Heidorn 1972, p.1].

8.5.6 Other Languages

8.5.6.1 The PL/I Branch

The cessation of support for GPSS V coincided with a decision by IBM to create the language SIMPL/I, a PL/I preprocessor [SIMPL/I 1972]. SIMPL/I has the distinct flavor of GPSS in its provision of the list processing, random number generation, and statistical routines, to assist in the modeling and simulation studies. Later, SIMPL/I and GPSS V commands were made interchangeable, with the translation of both into PL/I. Such a package, described as SIMPL/I X or PL/I GPSS, provided even greater flexibility to the modeler [Metz 1981]. Only a few IBM 360/370 Assembly Language routines augmented the PL/I implementation.

SIMPL is adescendent of OPS-4, created by Malcolm W. Jones and Richard C. Thurber [Jones 1971a, 1971b]. Implemented on the MULTICS time sharing system, SIMPL was envisioned as a simulation system consisting of both a programming language and a run-time support system, quite similar to its ancestors. Again, PL/I provided the underlying translation capability with SIMPL source code being compiled into PL/I. Use of SIMULATE (model name) set up the special environment for simulation and initiated the model. Like OPS-4, the interactive time sharing environment, now provided by MULTICS, served as the major vehicle.

A third member of the PL/I family, SIML/I, actually appeared in 1979 [MacDougall 1979]. Intended for computer system modeling, as was one of its predecessors ASPOL [MacDougall 1973], SIML/I provided representational capability “which extends into the gap between system-level and register-transfer-level simulation languages,” [MacDougall 1979, p. 39]. The process interaction world view is apparent, perhaps influenced more by SOL than SIMULA. Process communication occurs via signals that can be simple or synthesized into more complex expressions. The influence of the application domain is readily apparent in the representation of signals.

8.5.6.2 The Pritsker Family

A. Alan B. Pritsker and Pritsker and Associates, Incorporated have contributed significantly to the development of simulation languages. In addition to GASP II and subsequent versions, GERTS (GERT Simulation program) provided a network modeling capability for simulation solutions to activity network models and queueing network formulations based on GERT, created earlier by Pritsker. Extensions to the network modeling capability resulted in a family of programs generally identifiable as one form of GERT or another. The GERT family tree is shown in Figure 8.14 which also shows the relationship to GASP [Pritsker 1990, pp. 242-243].

SAINT (Systems Analysis for Integrated Networks of Tasks) was developed by Pritsker and Associates under an Air Force contract that sought a modeling capability that enabled the assessment of contributions of system components especially human operators to overall performance [Wortman 1977a]. SAINT utilized a graphical approach to modeling, similar to Q-GERT, and depended on an underlying network representation. The continuous component was similar to the provided in GASP IV [Wortman 1977a, p.532]. Like GASP IV, SAINT is actually a simulator consisting of callable FORTRAN sub-routines and requiring approximately 55,000 decimal words of internal storage [Wortman 1977b, 1977c]. The contributions of numerous students and employees are generously described in Pritsker [1990].
8.5.6.3 Interactive Simulation Languages

The rapid acceptance and increased availability of time-sharing operating systems led to the emergence of interactive modeling systems based on user dialogue or conversational style. Two such languages during this period are CML (Conversational Modeling Language) developed by Ronald E. Mills and Robert B. Fetter of Yale University [undated]. CML was intended for language creation as well as a language for simulation purposes. The reference describes the use of CML for creation of a simulation language used to study hospital resource utilization. The power of CML was found in its ability to provide deferred run-time definition of model parameters and its ability for model modification and redefinition. The creators describe CML as equally useful as a simulation language, a general purpose programming tool, a specialized “package” for certain tasks, and an environment for systems programming.

CONSIM was developed as a doctoral dissertation by Sallie Sheppard Nelson at the University of Pittsburgh. Considered a prototype conversational language, CONSIM is patterned after SIMULA 67 because of the recognized advanced capabilities of the language [Sheppard Nelson 1977, p. 16]. Coroutine sequencing, process description, and dynamic interaction in the CONSIM interpreter follow the design found in SIMULA 67.

8.6 CONSOLIDATION AND REGENERATION (1979–1986)

The period from 1979 to 1986 might be characterized as one in which predominant simulation languages extended their implementation to many computers and microprocessors while keeping the basic language capabilities relatively static. On the other hand, two major descendants of GASP (in a sense) appeared to play major roles: SLAM II and SIMAN.

8.6.1 Consolidation

Versions of GPSS on personal computers included GPSS/PC developed by Springer Cox and marketed by MINUTEMAN Software [Cox 1984] and a Motorola 68000 chip version of GPSS/H [Schriber 1984, p. 14]. Joining the mapping of GPSS to PL/I are a FORTRAN version [Schmidt 1980] and an APL version [IBM 1977].

CACI extended SIMSCRIPT II.5 even further by providing application dependent interfaces: NETWORK II.5 for distributed computing in September 1985 and SIMFACTORY II.5 for the modeling of manufacturing problems in October 1986 [Annino 1992]. An added interface in November 1987, COMNET II.5, addressed the modeling of wide and local area communications networks.

An extension of SIMULA, to capture an explicit transaction world view, called DEMOS, was introduced by Birtwistle [1979]. DEMOS was intended to provide modeling conveniences (built in resource types, tracing, report generation capability) that were lacking in SIMULA [Birtwistle 1981, p. 567].

8.6.2 SLAM AND SLAM II

The Simulation Language for Alternative Modeling (SLAM), a GASP descendent in the Pritsker and Associates, Inc. software line, sought to provide multiple modeling perspectives: process, event, or state variables, each of which could be utilized exclusively or joined in a combined model [Pritsker 1979]. SLAM appeared in 1979; SLAM II, in 1983 [O’Reilly 1983]. (Note that the acronym SLAM
is also used for a continuous simulation language developed in the mid-1970s [Wallington 1976]. The two are not related.)

8.6.2.1 Background
The background for SLAM introduction included the coding of processes as an addition to GASP IV in a version called GASPPI. This work was done as a master’s thesis by Ware Washam [Washam 1976a]. Jerry Sabuda in a master’s thesis at Purdue did the early animation work for Q-GERT networks that led to its incorporation eventually in the SLAM II PC animation program and SLAMSYSTEM® [Pritsker 1990, p. 292]. Pritsker [1990, p.290–293] describes the long, evolutionary process in developing highly usable simulation languages.

The intended purpose of SLAM was to join diverse modeling perspectives in a single language that would permit a large degree of flexibility as well as strong capability to deal with complex systems. Modeling “power” was considered the primary support capability influencing the design of the language and the later evolution of SLAMSYSTEM® [Pritsker 1991].

8.6.2.2 Rationale for Language Content
Clearly, GASP, Q-GERT, SAINT, and several variations of each contributed to the creation of SLAM. In fact, many of the identical subroutine and function names in GASP IV are repeated in SLAM.

8.6.2.3 Language Design and Definition
Unlike its predecessors which were simulators, SLAM is a FORTRAN preprocessor, with the preprocessing requirements limited to the network modeling perspective. The simulator aspect is preserved in the representation of continuous and discrete event models. Improvements were made to the functions and subroutines making up the discrete event and continuous components, but the major structure of the SLAM design followed that of GASP.

8.6.2.4 Nontechnical Influences
After the completion and delivery of SLAM as a product, differences occurred between Pritsker and Pegden over the rights to the language and a court case ensued. Settlement of the case dictated neither party should say more about the relationship of SLAM to SIMAN, a language developed in 1980–1983 by Pegden. The statements below are taken from Prefaces to the most current sources for each language.

C. Dennis Pegden led in the development of the original version of SLAM and did the initial conception, design, and implementation. Portions of SLAM were based on Pritsker and Associates’ proprietary software called GASP and Q-GERT. Since its original implementation, SLAM has been continually refined and enhanced by Pritsker and Associates [Pritsker 1986, p. viii].

Many of the concepts included in SIMAN are based on the previous work of other simulation language developers. Many of the basic ideas in the process-orientation of SIMAN can be traced back to the early work of Geoffrey Gordon at IBM who developed the original version of GPSS. Many of the basic ideas in the discrete-event portion of SIMAN can be traced back to the early work by Philip Kiviat at U.S. Steel who developed the original version of GASP. SIMAN also contains features which Pegden originally developed for SLAM. Some of the algorithms in SIMAN are based on work done by Pritsker and Associates. The combined discrete-continuous features of SIMAN are in part based on SLAM [Pegden 1990, p. xi].
8.6.3 SIMAN

The name SIMAN derives from SIMulation ANalysis for modeling combined discrete event and continuous systems. Originally couched in a manufacturing systems application domain, SIMAN possesses the general modeling capability for simulation found in languages such as GASP IV and SLAM II.

8.6.3.1 Background

Developed by C. Dennis Pegden, while a faculty member at Pennsylvania State University, SIMAN was essentially a one-person project. A Tektronix was used as the test bed for the language, which originated as an idea in 1979 and moved rapidly through initial specifications in 1980, to final specification and a prototype in 1981. A full version of the language was completed in 1982, and the release date was in 1983.

8.6.3.2 Rationale for Language Content

SIMAN incorporates multiple world views within a single language:

1. a process orientation utilizing a block diagram similar to that of GPSS,
2. an event orientation represented by a set of FORTRAN subroutines defining instantaneous state transitions, and
3. a continuous orientation, utilizing dependent variables representing changes in state over time (state variables).

Thus, SIMAN is either a FORTRAN preprocessor or a FORTRAN package depending on the selected world view. The use of FORTRAN as the base for SIMAN was justified by the wide availability of the latter language on mainframes and minicomputers. Pegden [1991] acknowledges that in today’s technical market place it would be much easier if written in C or C++.

Influencing Factors SIMAN draws concepts from GPSS, SIMSCRIPT, SLAM, GASP, and Q-GERT [Pegden 1991]. The primary contributions are the combined process, next event, and continuous orientation from SLAM and the block diagram and process interaction concepts from GPSS.

New concepts introduced in SIMAN include general purpose features embedded in specialized manufacturing terminology, for example, stations, conveyors, transporters. SIMAN has claimed to be the first major simulation language executable on the IBM PC and designed to run under MS-DOS constraints [Pegden 1991]. Macro submodels provide convenient repetition of a set of objects without replication of entire data structures.

Nontechnical Influences Systems Modeling Corporation in marketing SIMAN recognized the major advantage of output animation and created a “companion” product called CINEMA.

8.6.4 The PASCAL Packages

The emergence of yet another popular general purpose language—Pascal—stimulated a repetition of history in the subsequent appearance of simulation packages based on the language. Bryant [1980; 1981] developed SIMPAS as an event scheduling language through extensions of Pascal that met the six requirements for simulation (described in section 8.1.1.2). Implemented as a preprocessor,
SIMPAS was designed to be highly portable yet complete in its provision of services. In contrast, PASSIM [Uyeno 1980] provided less services, requiring more knowledge of Pascal by the modeler.

INTERACTIVE, described as a network simulation language, used graphical symbols in a four-stage model development process that supported interactive model construction and execution [Lakshmanan 1983]. The implementation on microcomputers, coupled with the ability of Pascal to support needed simulation capabilities, motivated the development of the package [Mourant 1983, p. 481].

8.6.5 Other Languages

INSIGHT (INS) was developed by Stephen D. Roberts to model health care problems and as a general simulation modeling language [Roberts 1983a]. Utilizing the world view of a network of processes, INSIGHT adopts the transaction flow characterization of object interaction with passive resources. Described as a simulation modeling language rather than a programming language or parameterized model [Roberts 1983b, p. 7], INSIGHT provides a graphical model representation that must be translated manually into INSIGHT statements. INSIGHT provides the usual statistical utilities for random number generation but also provides assistance for output analysis. A default output of model behavior can be supplemented through a TABLE statement, and the utilization of FORTRAN is always available for the user of this preprocessor.

8.7 CONCLUDING SUMMARY

The history of simulation programming languages is marked by commercial competition, far more intense than that of general purpose languages. Perhaps that single fact explains the identifiable periods of remarkably similar behavior. Such competition might also be the motivator for the numerous concepts and techniques that either originated with an SPL or gained visibility through their usage in the language. Among the most significant are:

- the process concept,
- object definition as a record datatype,
- definition and manipulation of sets of objects,
- implementation of the abstract data type concept,
- quasi-parallel processing using the coroutine concept,
- delayed binding with run-time value assignment,
- English-like syntactic statements to promote self-documentation,
- error detection and correction in compilation,
- dynamic storage allocation and reclaim, and
- tailored report generation capabilities.

Acknowledged as the first object-oriented programming language, SIMULA 67 with its combination of features such as encapsulation, inheritance, the class and coroutine concepts, still remains a mystery to the large majority of the programming language community. Moreover, it remains an unknown to the majority of those hailing the object-oriented paradigm, irrespective of their knowledge in discrete event simulation.
Although the ALGOL roots are often cited as the cause of SIMULA’s relative obscurity, what dooms SIMSCRIPT to a similar fate? Is it the FORTRAN roots? Addressing the issue more seriously, the entity-attribute-relational view of data was both implicitly and explicitly enunciated in SIMSCRIPT fully ten years before Peter Chen’s landmark paper in database theory [Chen 1976]. Yet, neither the database systems nor the programming languages community took notice.

Given the early prominence attached to simulation programming languages, reflected by the significance of meetings such as the IFIP Working Conference [Buxton 1968] and the eminence of the attendees, what has led to the appearance of lack of interest by the larger community (be it programming languages or computer science)? Has this subdisciplinary insularity been counter-productive in the development of current general purpose languages? Is this insularity a two-way phenomenon and, if so, is MODSIM (a recently developed object-oriented SPL) likely to suffer for it? Although a series of questions might be thought unseemly to close a paper such as this, the answers to them seem crucial to answering a more fundamental question of the programming language community:

Is the past only a prologue?

ACKNOWLEDGMENTS

I am deeply indebted to Philip J. Kiviat for his guidance throughout the development of this paper. I am grateful also to those who provided personal information on various languages and other topics: Joseph Annino, Nick Cooper, Donald E. Knuth, C. Dennis Pegden, A. Alan B. Pfitzker, and Paul F. Roth. My thanks also to my colleagues, James D. Arthur, for his help in language and translator characteristics, and Osman Balci, for discussions of language and model representational issues.

REFERENCES

PAPER: A HISTORY OF DISCRETE EVENT SIMULATION PROGRAMMING LANGUAGES


DISCRETE EVENT SIMULATION LANGUAGE SESSION

409
RICHARD E. NANCE


RICHARD E. NANCE

DICK NANCE: I’m going to stand a little closer to the projector and carry you through a number of slides fairly rapidly. I encourage you to read the paper, since obviously I am not covering anywhere near the extent of the material that’s provided in the paper. I’m skating, and pulling material that appears to me to be perhaps the most interesting to you.

(SLIDE 1) I’ll talk about the technical background very briefly, give you some idea of the historical perspective, the organization of the talk, and how periods arose as a way of organizing the material here. In other words, I’ve taken a horizontal cut in time, in talking about all simulation languages during that horizontal period. The reason for it is that they went through a number of very similar experiences, and then there will be selected language sketches; I cannot talk about all the languages. I will pick and choose just a few, and then give a concluding summary.

(SLIDE 2) In terms of simulation, I think the term is often associated with three different forms: discrete event simulation portrays state changes at precise points in simulated time, continuous simulation is typically thought of as state equations for solutions of differential equations or equational models, and Monte Carlo simulation, which is a sequence of repetitive trials in which you have a very simple state representation, nothing like the complexity you can have in a discrete event model. And then there are, as I have shown across the bottom, three sorts of different forms based on how you view the relationship with the three above. A hybrid model will fall within a discrete event simulation model, typically. Combined simulation deals with combined discrete event and continuous modeling, and then gaming is an area where simulation is very crucial, but there are other factors too that need
(SLIDE 3) To be considered. In the red or magenta, are shown the two forms that I will concentrate on, both here and in the paper, the discrete event and the combined.

(Slide 3) There are three different perspectives that I advance in the paper for looking at discrete event simulation and languages for discrete event simulation: the requirements for a language, the different forms for a language, and then something that is very important to people working in the field of discrete event simulation, the conceptual framework that a language supports.

(SLIDE 4) Each simulation language must meet these six requirements: a means of random number generation, some way of transforming the random numbers (from the U(0,1) to a more general set of distributions), a capability for list processing (creating, deleting, and manipulating objects), a basis for statistical analysis of output, some means of report generation (so that behavior can be characterized), and then a timing executive or a time-flow mechanism.

(SLIDE 5) In terms of language forms, we find simulation packages being a very predominant form. Now I know that a language purist doesn’t view a package as a language per se, but I had to include them here, because they have had such a major effect historically. GSP, the general simulation program, GASP, the general activities simulation program, and the Control and Simulation Language—three packages from the very early days that, in fact, have influenced—and continue to influence—languages today.

Pre-processors are another form, and two of the current languages, SLAM, Simulation Language for Alternative Modeling and SIMAN simulation analysis language, represent preprocessors. Then the more typical programming languages themselves, GPSS (was represented at the first conference by Geoffrey Gordon), Simscript II from the Rand Corporation—Phil Kiviat and Harry Markowitz—you’ve already heard Harry Markowitz mentioned; you’ll hear Phil Kiviat mentioned a great deal. And in fact, at this point, let me say, that I am in debt to Phil Kiviat for his help in preparing both the paper and the presentation. Phil is a tremendous resource and a great individual to work with. And then SIMULA, which needs no introduction—you’ve already heard several people in the conference and preceding speakers discuss the effect SIMULA has had in areas like LISP, and Per Brinch Hansen noted its effect in operating systems and Concurrent Pascal.

(SLIDE 6) To explain conceptual frameworks, this third perspective, let me start with a very simple example of a type of model that is used in discrete event simulation, a single server queuing model—if you like to think in applications, think of a single barber in a barber shop. In such a model, you will have a process model of jobs or customers arriving and a queue formed if the server is busy. You will have a first-come first-served order to the service, and a server idle if that queue is empty. The objective
TRANSCRIPT OF SIMULATION PRESENTATION

Language Forms

- Simulation packages:
  GSP (Tocher), GASP (Kiviat), CSL (Buxton and Laski)
- Preprocessors:
  SLAM (Peters), SIMAN (Pegden)
- Simulation Programming Languages (SPLs):
  GPSS (Gordon), SIMSCRIPT II (Kiviat and Markowitz), SIMULA (Dahl and Nygaard)

Conceptual Frameworks

Time and state characterization: single-server queue model

<table>
<thead>
<tr>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 1 Complete</th>
<th>Job 2 Complete</th>
<th>Job 3 Complete</th>
</tr>
</thead>
</table>

There may be to compute the job wait versus the server idle time, because they represent costs that you are trying to balance. Now, the views that are exemplified by the time and state characterizations of such a model, might be when events occur, when jobs arrive, or when services end.

(SLIDE 7) This has been characterized by Mike Overstreet as temporal locality and the US refers to where it geographically was considered the way to describe a system: in this case, the United States. Event scheduling or next event languages were typical of the US view of how to do simulation modeling.

Activity duration, where you concentrated on the state of the simulation over some period of time, was more popular in the United Kingdom—there you might look at the state of a model, of a job waiting, or a job in service, for the period where the server is idle.

The third framework, the object existence, the encapsulation here of locality of object as Overstreet characterized it, in contrast with locality of state or locality of time. You look at all that occurs with respect to the job; its entire experience as it goes through a system, as we have described above.

(SLIDE 8) Now, the event scheduling conceptual framework focused on the epochs marking state change. What happens or what are the consequences of a job arriving? Well, I have to describe them as “what is the next thing that will happen to this job—it will be served if the server is available?” But, you notice here, there are points in time which are pulled out for emphasis in the description;
those points in time relate to the job, very obviously, but also to something not so obviously and that is the server.

(SLIDE 9) The activity-scan or class of languages concentrated on state, and described the state duration for the separation of the two events that either initiated or terminated the two state activities: job one arrives, and job one completes, but in between that time, job two arrives and must wait until the server is available.

(SLIDE 10) Now in the process interaction, the third form described as locality of object, we really have two subforms—the first process interaction or transaction view, as it was characterized in GPSS, was that you looked at the temporary objects, the customers or jobs, as they came into the system—you focused on what happened to that job. But behind the scenes, was the server, the permanent object, which is not well characterized.

(SLIDE 11) In fact, the problem, or the difference between the transaction view and the more traditional process interaction view, is the explicit representation of the server as a process as well as jobs as processes. Now you're characterizing what's happening with the server and jobs, and the points where they interact.

(SLIDE 12) What are some of the influences and consequences we see for having these differing conceptual frameworks that underlay the whole development of simulation languages? Well, the conceptual frameworks sort of created conceptual communities in and of themselves, and as we heard our friends talk about LISP, these communities were equally isolated—they didn't talk to each other much. Phil Kiviat characterized this, in his Rand Report of 1967, as an inversion of theory and interpretation. Where the language should be an interpretation of the theory, it became the theory, and so people working in SIMULA didn't talk to people working in SIMSCRIPT II. On the other hand, there was knowledge of the work going on, in many cases. I would say that the people working on the languages had far more knowledge of the other language forms than the people actually using the languages. There were differences because of application domains, and the frameworks themselves were at least stimulated by the application domains. And then there was a contrast in educational and industrial desires and constraints. Many simulation languages were developed because of a desire to use them in an educational setting. They had very different constraints and very different objectives from those developed to use in a business or industrial setting.

The consequence, of course, is that we ended up with numerous simulation languages—however you wish to define "language." In 1966–1967, two reviews at that time identified 21 different discrete event simulation languages—that does not include the continuous (simulation) languages. In 1981,
John Crooks said that he could count 137 different simulation programming languages (SPLs). If the estimate of a thousand programming languages is correct, then SPLs number over 10 percent—that's a very healthy number. Now, I'm sure that many of these are dead, and some of those that aren't perhaps should be.

(SLIDE 13) When I started preparing for this talk, I thought about different ways I could present the material. I could talk about, for instance, the different languages within a framework family. What were the activity scan languages? What were the event scheduling languages, etc.

(SLIDE 14) I could also talk about a genealogy of a particular language following GPSS or SIMULA or SIMSCRIPT and follow it through its rather interesting history of spinning off and creating others.

(SLIDE 15) But I chose a different way, I chose probably a horizontal perspective, where I cut across in periods of time and I did that because there are very interesting similarities that occur in all the languages during that period of time.

I think it is interesting to ask why do they occur? What is driving that?

(SLIDE 16) So there are the five periods that I will talk about.

(SLIDE 17) The period of search. That was the period prior to the existence of any simulation language, per se, from roughly 1955 to 1960. There were new general purpose languages appearing in that period of time, and there were some simulation models that were done in assembly language
RICHARD E. NANCE

Historical Organization: Horizontal Development by Period

Periods of Partitioning

- The Period of Search (1955–1960)
- The Advent (1961–1965)
- The Expansion Period (1971–1978)
- Consolidation and Regeneration (1979–1986)

Advantages:
- Clear picture of inter-family relationships
- Striking similarities

Disadvantages:
- Less than "natural"
- More difficult, even argumentative

The Period of Search (1955–1960)

- Programming in assembly languages
- Appearance of new general purpose languages (GPLs), e.g., FORTRAN
- Specific models in GPLs
  - General Electric
  - UCLA: Jackson, Nelson, Rowe
- General Simulation Program (GSP): K.D. Tocher, D.G. Owen (1958)

SLIDE 15

SLIDE 16

SLIDE 17

SLIDE 18

and the new general purpose languages. General Electric was one place, and the study of scheduling rules by Jackson, Nelson, and Rowe at UCLA around the 1958 or 1959 time period.

Then, around 1958, Tocher and Owen in the UK published a paper on the General Simulation Program.

(SLIDE 18) The General Simulation Program was essentially a search for a simulation structure. K.D. Tocher realized when he did programming of a simulation model he repeatedly did several things. He developed an initialization form or an initialization routine. He developed the description of what he called state-based activities or events and time-based events. Then he also had a report generator. So what he began to do, as every programmer does, is to create his set of routines which he reuses as he programs later simulation models. This became known as the General Simulation Program.

(SLIDE 19) When we move into The Advent (of real languages), we find some interesting things, at least I find them interesting, for today. For instance, this continuing legacy. All the major simulation programming languages of today can trace their roots back to the languages developed in the period of 1960 to 1965. With Control and Simulation Language, it was an activity scan language, a FORTRAN preprocessor and graphical input was provided for. One drew a picture of the model. GASP was an event scheduling language, also a FORTRAN package, and again a graphical interface was provided. GPSS was a process interaction, transaction language, interpretive in terms of its
transcription and was also graphical. SIMSCRIPT, event scheduling, again a FORTRAN preprocessor and textual in nature, no graphical interface. SIMULA, the process interaction ALGOL extension, is textual. There were others: DYNAMO and OPS-3.

You might ask why are you including DYNAMO? DYNAMO was not a discrete event simulation language. That’s true. But according to the persons who have contributed so much to the development of languages that are discrete event, they go back to claim many interesting influences, many interesting discussions with the developers of DYNAMO, and also, that they watched very carefully what was happening at MIT during the period of DYNAMO’s development.

(SLIDE 20) In trying to provide a language sketch, I both apologize and ask your understanding about the brevity of the sketches. I’ll try to identify the creators or developers of each language that I include, the organization or organizations here, contributors to the language, the predecessors or influencers, that is, preceding language or language versions. In some cases information about the level of effort is included, although it is not easy to pull that out. And then, any distinctive characteristics or points about the language.

(SLIDE 21) Control and Simulation Language (CSL), originating in 1962, John Buxton and John Laski were the developers. It was a joint development of Esso Petroleum and IBM UK. They cite Tocher and Hartley in Cambridge and Cromar of IBM as persons influential on them in the development of the language. The predecessors include Tocher’s GSP, Montecode, and FORTRAN—Montecode being a simulation model that was developed in the 1960 time frame.

CSL eventually became a compiled language, taking about nine staff-months in the development of the language. I could find no indication of how much effort was devoted to design. The second version of the language in 1965, called by some, C.S.L.2, provided dynamic checking of array subscripts, which I thought was an interesting capability at that point in time.

(SLIDE 22) GASP, the General Activity Simulation Program, began in 1961 with Philip Kiviat’s arrival at US Steel. It was basically a one-person effort although Belkin and Rao authored the GASP Users Manual, we think around 1965. It was done as a FORTRAN or FORTRAN package and had a lot of influence from the steel-making application domain. It took approximately 12 staff-months for both design and code, and was viewed principally as a problem-solving tool, not really a language for doing simulation. For instance, it had a regression model of input that was very capable for that period of time. It indicated an understanding of dependencies in the input stream, which you don’t find in other languages of that period.
**CSL: Control and Simulation Language**

- John N. Buxton and John G. Laski (1962)
- Jointly: Esso Petroleum + IBM U.K.
- K.D. Tocher, D.F. Hartley (Cambridge), L.J. Cromer (IBM)
- GSP, Montecode, FORTRAN
- Compiler: 9 staff-months; Design: ?

**GASP: General Activity Simulation Program**

- Philip J. Kiviat
- United States Steel
- FORTRAN, Application domain
- Approximately 12 staff-months: Design + Code
- Problem-solving focus, regression model of input

---

(SLIDE 23) GPSS is described in detail in HOPL I, so I include it here only for completeness. (SLIDE 24) SIMSCRIPT, whose conceptual father was Harry Markowitz, a Nobel Laureate in Economics, was based on the entity, attribute, and set descriptors. Much of what is later included in the entity-relationship model is found in SIMSCRIPT.

(SLIDE 25) SIMULA, Dahl and Nygaard, 1961—and as those of you who have read the proceedings from HOPL I know, this has had a major influence on simulation programming languages, general purposes languages, and computer science. I would like to say that if there is any question about it, as far as I am concerned—and I think I could defend this—SIMULA was the first and is the first object-oriented language. It was developed at the Norwegian Computing Center. Jan Garwick, the father of Norwegian computing, is identified as a major influence.

SIMSCRIPT and ALGOL, of course, were mutual influencers in their own right. Bernard Hausner, who was one of the early developers of SIMSCRIPT 1.5—in fact, he was the chief programmer—actually joined the team in 1963 at the Norwegian Computing Center. As the report in the 1981 HOPL proceedings indicate, Dahl and Nygard suddenly discovered the event scheduling view of the world that they had never even thought about. You notice the effort here is considerable, 68 staff-months, and this was just for SIMULA I. Key here are the process and code routine concepts. You immediately say, “what about the inheritance, what about class?” That really came with SIMULA 67 about four years later.

---

**GPSS: General Purpose System Simulation**

- Geoffrey Gordon (1960)
- IBM
- R. Barbieri, R. Fieron
- Sequence Diagram Simulator (D.L. Dietmeyer), Gordon Simulator
- Approximately 24 staff-months
- Block diagram, Interpreter, GPSS II, GPSS III

**SIMSCRIPT**

- Harry Markowitz, Bernard Hausner, Herbert Karr
- U.S. Air Force Project RAND
- R. Conway (Cornell)
- GEMS (Morton Allen), SPS-I (Jack Little)
- Approximately 36 staff-months
- Entity/Attribute/Set Description
(SLIDE 26) OPS-3, a very interesting language, not because of the influence it had, but because of what it set as goals for the future. OPS-3, developed at MIT, Martin Greenburger and Mal Jones, major influences there, at the Sloan School of Management. I always found it referred to as the "On-Line Programming System," however, Jean Sammet in her book in 1969 indicated that it was actually the "On-Line Process Synthesizer" and that is why I have shown that title in parentheses.

Using CTSS, the creators of OPS-3 showed us what a language and a capability for simulation could look like in a kind of time-sharing domain that few had at their disposal at that time. I cannot say much about the level of effort. The distinctive point here is that OPS-3 was considered a system not a language. It was a system for doing interactive development and model execution. As such, it really pushed into a domain that sort of defined a path or identified a path.

(SLIDE 27) To summarize: In the period of the advent, there were numerous simulation programming languages other than the ones that I have sketched—SIMPAC at SDC, Simulation Oriented Language, that Knuth and McNeley created. There were also several interesting comparison papers; I note seven papers during that period of 1963 to 1967. People were interested in what others were doing; what alternatives were being taken—I think in part because the simulation language and simulation modeling was such an intensive effort, both in development and in execution. There were many workshops, and Dan Teichrow—at Case at the time, and John Lubin at Pennsylvania were called by Kiviat "the honest brokers," the people who did not have vested interests in language but kept trying to move along the whole technology of language development. There was an IBM symposium in 1964 on simulation models and gaming. Tocher and Geoff Gordon spoke at that. There were several conferences. The 1967 IFIP Working Conference on SPLs, Doug Ross was at least one person there. There could be others here that attended that conference. It read like a "who's who" of personages in computer science. Tony Hoare was there. An interesting anecdote from this conference asked what were the plans for SOL. Knuth said, "We have no plans for SOL, I found everything save the wait-on-state condition that is provided in SOL provided in SIMULA, and I would encourage you to go to SIMULA."

In 1967, there also was the conference entitled Applications of Simulation Using GPSS that later expanded into Simulation Applications and then finally the Winter Simulation Conference, that's what WSC stands for. And today it remains as the premier conference for discrete event simulation.

(SLIDE 28) During the next period, which I have called the Formative Period, we find all the languages beginning to really focus on their concepts—to revise and refine these concepts and set forth what really distinguished that language from others. The Extended Control and Simulation
RICHARD E. NANCE

The Advent (1960–1965)

- Numerous SPLs: SIMPAC (SDC); SOL (Knuth & McNeley)
- Language Comparisons: Seven papers (1963–1967)
- Conferences and Workshops
  - Workshops (1964–1965): D. Teichroew, J. Lubin
  - Symposia: IBM — Simulation Models and Gaming (1964)
  - Conferences:
    - IFIP Working Conference on SPLs (22–26 May 1967)
    - Applications of Simulation Using GPSS (November 1967)

SLIDE 27


- Concept Extraction, Revision, Refinement
  - ECSL (Clementson):
    - FORTRAN abandoned
    - CAPS (Computer Aided Programming System)
  - GASP II (Pritsker and Kiviat):
    - Target: Small computers
  - GPSS/360 and GPSS V:
    - Expanded set operations
    - External access (HELP)
    - Free-format coding

SLIDE 28

Language (ECSL), now Allen Clementson became the major mover and FORTRAN was abandoned. A front end, which was called the Computer Aided Programming System (CAPS), was added. GASP II by Alan Pritsker and Phil Kiviat—really Alan Pritsker—becoming the mover and its target was simulation capability on small computers. GPSS-360 and then GPSS-V expanded the set operations, provided the external access to other languages through HELP, and then provided free-format coding.

(SLIDE 29) In concept extraction, revision, and refinement, we see that SIMULA now includes the object and class concept and the virtual concept. SIMSCRIPT II comes into being and OPS-4 provides a PL/I based language with all three world views.

(SLIDE 30) During this period, no language had really more effect than SIMSCRIPT. SIMSCRIPT II by Kiviat, Markowitz, and Villanueva at RAND became a prime competitor with GPSS. It took approximately 150 staff-months. It was interesting in that it had five different levels and an English-like forgiving compiler.

(SLIDE 31) Other languages, the Burroughs Operational System Simulator (BOSS), Q-GERT, a network modeling language proposed by Pritsker, and SIMPL/I, a language from IBM based on PL/I.

(SLIDE 32) During 1971 to 1978 we find the period of extensions and enhancements (Period of Expansion). Now languages previously described in terms to distinguish them began to effect a blurring of these distinctions. GASP from Pritsker emerged as GASP IV in 1974, which had the capability for doing combined discrete event/continuous modeling, and then also provided a process...
The Formative Period (1966-1970)

- Other Languages
  - BOSS: Burroughs Operational System Simulator (1967)
    A.J. Meyerhoff, P.J. Roth, P.E. Shafer
  - Q-GERT: Network-modeling (1970)
    A.B. Pritsker
  - SIMPL/I: IBM, Based on PL/I (1967)
    N. Cooper, K. Blake, U. Gross

The Expansion Period (1971-1978)

- Extensions and Enhancements
  - GASP (Pritsker)
  - SIMPL/I: IBM, Based on PL/I (1967)
    N. Cooper, K. Blake, U. Gross
  - GPSS
    NGPSS and NORDEN: J. Reitman (1971-73)
    Interactive, Animation (crude)
    Numerous versions: GPSS V-6000, GPDS,
    GPSS 1100, GPSS/UCC
    GPSS-H: Compiled version

 interaction—that is, GASP provided a vehicle to look much like GPSS in terms of its view of the world. GPSS took a different form in the Norden GPSS and GPSS/NORDEN versions, Julian Reitman being a major mover here. It actually had interactive capability, and provided some basis for rather crude animation to look at the output. Then there were other versions of GPSS.

(SLIDE 33) During this period, SIMSCRIPT II.5 develops a combined view for discrete event/continuous models and also adopts a transaction view like GPSS. And then extensions for computer system modeling emerge. SIMULA 67 really moves away from the machine and toward the user with this idea of a pure system description language called Delta.

(SLIDE 34) Then ECSL moves into program generators. Program generators are an interesting concept—they actually began at RAND in 1965, but interactive generators in the UK became the major means of creating models. They were based on the original Tocher Wheel Charts, but became well known as activity or entity cycle diagrams. They were not intended to provide a complete model description but a draft. And, in fact, that is what Matheson at Imperial College called his generator DRAFT of a model that can then be transformed into different implementation forms.

(SLIDE 35) This is just one example of a single server queue shown as an entity cycle diagram, where you see jobs arriving from the world, going into a queue, then meeting the server in the period of service, and then going back to the world. This became a way of describing a model. It could be automatically translated into a draft or outline of an executable program.
We also had interactive simulation languages, the Conversational Modeling Language from Yale and the Conversational Simulation Language by Sallie Sheppard at Pittsburgh based on SIMULA 67.

Other simulation languages were created; I have identified some of them here. You notice that SIMPL is a descendent of OPS-4. SIML/I, by MacDougall was primarily for computer system modeling. Then there was a family of languages introduced by Pritsker Corporation or Pritsker and Associates that included the GERTS language and the SAINT languages—Systems Analysis for Integrated Systems of Tasks (I think that is what SAINT stood for).

In '79 to '86 we had the period that I called the Consolidation and Regeneration. Now we have different versions of GPSS appearing on personal computers; we have a GPSS FORTRAN in Germany, and we have a GPSS/APL. Did you know that there was such a beast? We had different application domain dialects being developed for SIMSCRIPT II.5, SIMFACTORY II.5, NETWORK II.5, and COMNET II.5. There is even a LanNet II.5. Then with SIMULA 67, we have an extension DEMOS that makes it look again like GPSS, in that it takes a transaction world view which is viewed as a much simpler approach for teaching.

In terms of regeneration we find two languages, SLAM and SIMAN. Of interest here is that the people creating these two languages worked together, and then worked separately and you
### Consolidation and Regeneration (1979–1986)

- **Regeneration**
  - SLAM (Pritsker Corporation): Simulation Language for Alternative Modeling
    - SLAM (1979) and SLAM II (1983)
  - SIMAN (Systems Modeling Corporation): SIMulation ANalysis
    - Initial idea: 1979
    - Released: 1983
  - Litigation

### SLAM and SLAM II

- A.A.B. Pritsker and C.D. Pegden
- Pritsker and Associates, Inc.
- J. Sabuda, W. Washam
- GASP, Q-GERT, SAINT
- Effort through GASP evolution (graduate students)
- FORTRAN preprocessor, Graphical, Multiple world views, Combined simulation

---

**SLIDE 39**

notice the third bullet. You can read in the paper what is said about the result of the court suit between SIMAN and SLAM.

(SLIDE 40) SLAM developed by Prisker and Pegden, was a natural evolution of GASP. It became a preprocessor though, as opposed to a language package. It consolidated a lot of things developed by Prisker’s graduate students. It still is a FORTRAN preprocessor, using a graphical input, and providing multiple world views and combined simulation.

(SLIDE 41) Pegden’s SIMAN, was a faculty project at Penn State—really a single person effort that took approximately 24 staff-months over a four-year period. It drew from a lot of different languages but I would have to say notably from SLAM since Pegden was a codeveloper there. It was the first major simulation programming language on a PC. It showed the PC as a viable platform. It had manufacturing applications features and provided a CINEMA animation report capability.

(SLIDE 42) In conclusion I find that simulation programming language history is characterized by a number of things. First of all, a large number of languages—long lived major languages, those that have been with us for thirty years—and very spirited commercial competition. Pick up a copy of *Industrial Engineering* and note the number of advertisements for simulation software.

In part, this spirited competition contributed to the similarities in SPL development during the recognized periods used for organizational division. While the advent of a new general purpose language, such as Pascal or C, typically led to simulation packages in that language, a mutual

### SIMAN

- C. Dennis Pegden (1979–1983)
- Faculty project (Penn State)
- Single person
- GPSS, SIMSCRIPT, SLAM, GASP, Q-GERT
- Approximately 24 staff-months (over four-year period)
- FORTRAN preprocessor, Graphical, Multiple world views, Combined simulation, CINEMA animation

### Concluding Summary

- SPL history characterized by
  - Large number of languages
  - Long-lived major languages
  - Spirited commercial competition
  - Periods of comparatively similar development
  - Progressive mutual disinterest

(Simulation ≠ Programming languages)
indifference has characterized the relations between the two language communities. Hopefully the recognition of programming as a form of modeling and the evolution of model development environments will produce a more symbiotic relationship.

**TRANSCRIPT OF QUESTION AND ANSWER SESSION**

**HERBERT KLAEREN (Univ. of Tubingen):** It seems to me that your six requirements for a discrete event simulation language can be fulfilled using a library for any general purpose programming language. What are the drawbacks of the “library approach” that justify the development of dedicated simulation languages?

**NANCE:** Perhaps my paper makes it clearer than the presentation that packages in GPLs have always been major contributors, and my conjecture is that, during the period covered (1955–1986) more simulation models have been developed in GPLs than in SPLs. However, there are major advantages to using an SPL: (1) statistics collection can be made much simpler for the user, (2) the user interface can accommodate nonprogrammers more readily, and (3) application domain knowledge can be incorporated, or the provisions made for doing so, with less difficulty.

**ROSS HAMILTON (Univ. of Warwick):** Your categorization of periods in the development of simulation languages seems to infer that they have “evolved” and that they have now completed their evolution. Would you please care to comment on this?

**NANCE:** That inference is certainly not intended. However, my perhaps biased (by my own research interests) view is that current emphasis is on model specification at a more abstract level, with transformation to an execution specification that remains hidden from the user. This emphasis follows the “automation based paradigm” of [Balzer 1983], and attempts to move the model development responsibility to users more expert in the application domain that are not programmers. Nevertheless, problems remain that motivate new implementation concepts, and specification techniques will be offered commercially that are tied to existing SPLs or their progeny. As I note in my paper, developments after 1986 are so recent as to defy a historical perspective.

**CHARLES LINDSEY (U. of Manchester):** Has there been any standardization activity in the field (ANSI or ISO), and would such be desirable?

**NANCE:** Yes, there is a standard for Simula67, and those standards activities are well managed from what I see. GPSS/H and Simscript II.5 are proprietary and have active user groups. For other SPLs, whether increased standardization would be welcomed and by whom is not clear.

**JAN RUNE HOLMEVIK (Univ. of Trondheim):** Do you see any difficulties in explaining historical developments genealogically, like in Figure 8.1 on page 373 of your paper?

**NANCE:** First, I would not characterize the genealogical tree as an “explanation” of historical developments. The tree shows chronological relationships that furnish a perspective on a large body of work. Clearly, an understanding of historical developments requires far more in the way of textual and graphical material; else, I might have supplied you with a paper containing a single figure.

**TOM MARLOWE (Seton Hall Univ.):** Does the move to make symbolic languages work in a standard interface/environment, such as Windows, address your concerns about language extension in any way (at least as far as the issues of familiarity and “front-end”)?
BIOGRAPHY OF RICHARD E. NANCE

NANCE: Most definitely, and my colleague Osman Balci and I have been working for the last ten years on research in model development environments. Specification of model behavior must be done at a higher level of abstraction than the implementation level. Further, our expectations are producing models far larger and more complex than we can hope to deal with successfully unless we can call upon automated support for diagnosis of specifications for correctness, component reuse at both design and program levels, etc. You might wish to examine a summary paper that describes this work [Balci 1992].

REFERENCES (for Questions and Answers)

BIOGRAPHY OF RICHARD E. NANCE

Richard E. Nance is the RADM John Adolphus Dahlgren Professor of Computer Science and the Director of the Systems Research Center at Virginia Polytechnic Institute and State University. He received B.S. and M.S. degrees from N.C. State University in 1962 and 1966, the Ph.D. degree from Purdue University in 1968. He has served on the faculties of Southern Methodist University and Virginia Tech, where he was Department Head of Computer Science, 1973–1979. Dr. Nance has held research appointments at the Naval Surface Weapons Center and at the Imperial College of Science and Technology (UK). Within the Association for Computing Machinery (ACM), he has chaired two special interest groups: Information Retrieval (SIGIR), 1970–1971 and Simulation (SIGSIM), 1983–1985. He has served as Chair of the External Activities Board and several ACM committees. The author of over 100 papers on discrete event simulation, performance modeling and evaluation, computer networks, and software engineering, Dr. Nance has served on the Editorial Panel of *Communications ACM* for research contributions in simulation and statistical computing, 1985–1989, as Area Editor for Computational Structures and Techniques of *Operations Research*, 1978–1982, and as Department Editor for Simulation, Automation, and Information Systems of *IEEE Transactions*, 1976–1981. He served as Area Editor for Simulation, 1987–1989 and as a member of the Advisory Board, 1989–1992, *ORSA Journal on Computing*. He is the founding Editor-in-Chief of the ACM *Transactions on Modeling and Computer Simulation* and currently serves as the US representative to TC-7: System Modeling and Optimization of the International Federation for Information Processing. He served as Program Chair for the 1990 Winter Simulation Conference. Dr. Nance received a Distinguished Service Award from the TIMS College on Simulation in 1987 and was inducted as an ACM Fellow in 1996. He is a member of Sigma Xi, Alpha Pi Mu, Upsilon Pi Epsilon, ACM, IIE, and INFORMS.