Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

COMPLEXITY OF SIMULATION MODELS: A GRAPH THEORETIC APPROACH

Lee Schruben
School of Operations Research
and Industrial Engineering
Cornell University
Ithaca, New York 14853
USA

Enver Yücesan
European Institute of Business Admn
INSEAD
Boulevard de Constance
77305 Fontainebleau Cedex
France

ABSTRACT

Complexity of a simulation model is defined as a measure that reflects the requirements imposed by models on computational resources. It is often related to the structural properties of models. In this paper, we introduce complexity measures for simulation models using the concept of Simulation Graphs. A reasonable measure of complexity is useful in a priori evaluation of proposed simulation studies that must be completed within a specified budget. They can also be useful in classifying simulation models in order to obtain a thorough test bed of models to be used in simulation methodology research. Some surrogate measures of run time complexity are also developed.

1 MOTIVATION

A formalism provides a set of conventions for specifying a class of objects in a precise, unambiguous, and paradigm-free manner. The structure of a formalism further provides a basis for measuring the complexity of objects in that class. For example, complexity measures for *finite-state machines* include number of inputs, number of states, and number of outputs; number of vertices, number of directed edges, maximum fan-in, and maximum fan-out are some of the complexity measures for *directed graphs*. Analogously, complexity of a simulation model is defined as a measure that reflects the requirements imposed by models on computational resources. As Zeigler [1976; p.31] points out, these requirements can be characterized in various different forms:

1. the space the program requires in the computer to represent the model structure,
2. the time it takes to simulate one realization of the model,
3. the time and effort involved in constructing, implementing, testing, modifying, maintaining, and communicating the model.

Ultimately, the complexity of a model can be measured in terms of the extent of the resources it needs for development and simulation; this complexity is often related to the structural properties of the model [Zeigler, 1976].

The objective of this paper is to propose *complexity measures* that are directly related to the structural properties of simulation models. The main focus is on implementation and maintenance costs (as specified in 1 and 3); the execution time (as defined in 2) is of secondary interest. Simulation Graph Models, are introduced to assess the structural properties of models. The complexity measures are then defined with respect to these properties.

An early attempt to define the complexity of simulation models is by Evans, Wallace and Sutherland [1967; p.133]. Their measures include the number of different types of units (entities) in the model, the interaction among units, the proportion of "contingent" events, and the number and complexity of decisions made by event routines. These measures are principally abstract concepts as the authors provide no concrete metrics to quantify them.

A practical measure of complexity for simulation models will find ready utility in a priori evaluation of simulation models. This would be important for sponsors of large simulation projects, because a model whose complexity exceeds the limits imposed by the project budget is useless even if it is valid within some experimental framework. In particular, these measures will provide a quantitative basis for modularization that would allow program developers to identify software modules that will be difficult to test or maintain.

A perhaps more important application of simulation model complexity measures could be in evaluating simulation modeling and analysis techniques. The major problem in simulation methodology evaluation is in selecting a reasonable test bed of simulation models. Even a partial ordering of the complexity of simulations would help insure that the selected test models cover a wide range of possibilities.

This paper is organized as follows: §2 introduces and briefly discusses Simulation Graph Models. §3 discusses some intuitive measures of model complexity. In §4, a complexity measure based on the cyclomatic number of a connected graph is introduced. Illustrations of these measures are presented in §5. A surrogate measure for run time complexity is discussed in §6. Concluding remarks are presented in §7.

2 SIMULATION GRAPH MODELS

The elements of a discrete event simulation are *state variables* that describe the state of the system, *events* that alter the values of state variables, and the *logical* and *temporal relationships* between events. An *event*

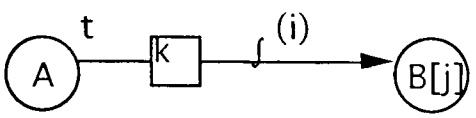
graph [Schruben, 1983] is a structure, a network, of these elements in a discrete event system that facilitates the development of a correct simulation model. Within this representation, the emphasis is directly on system events; system entities are represented implicitly as event attributes.

Events are depicted on the network as vertices (nodes). Typically, events are programmed as separate routines or procedures in an event-scheduling simulation. These event routines may transform state variables, generate delay times, test event incidence conditions, and perhaps schedule or cancel further events. Each vertex is associated with a set of changes to state variables. These variables are used to describe system entities. State variables can be thought of as event conditioning, performance monitoring or both.

Relationships between events are represented in an event graph as directed edges (arcs) between pairs of vertices. Each edge is associated with sets of logical and temporal expressions. Two types of edges are distinguished. *Scheduling* edges appear as solid arcs on the graph while *cancelling* edges are depicted as dashed arcs. Basically, the edges define under what conditions and after how long of a time delay an event will schedule or cancel another event. There can be multiple edges between any pair of vertices; the edges can point in either direction or may simply point from a vertex to itself.

In an event graph, it is also possible to parameterize the event vertices. Event parameterization is a modeling convenience and does not augment the modeling capabilities of these graphs. Parameters simply keep the graphs from being cluttered or possibly infinite [Yücesan, 1990]. This is achieved through *vertex parameters* and *edge attributes*. A *vertex parameter list* is a string of state variables associated with a particular vertex. An *edge attribute list*, on the other hand, is a string of expressions associated with a particular edge. These lists are used in scheduling or cancelling specific instances of events. For example, in a simulation model depicting the operation of a group of machines, a single *start_processing* event together with parameters can be used to model the start of processing on any of the machines in the group. This practice is analogous to passing values to a subroutine using a list of arguments in a high level programming language.

In summary, the construct,



is interpreted as follows: *whenever event A occurs, if condition (i) holds, event B is scheduled to occur in t time units with the parameter string, j, assuming the value k.*

The major advantage of this modeling framework is that the network provides a complete and consistent environment for model development. No new icons or constructs need to be defined as the modeling requirements change. In addition, this framework offers practical guidance not only for *model specification*, but also for *model implementation*. For instance, the customary notion of an event routine (see, for example Simscript [CACI, 1976]) in a simulation program may correspond to a subgraph, typically a set of vertices connected by edges with no delays. Alternatively, these graphs can be implemented directly using SIGMA [Schruben, 1991] either to be executed in an interpretive fashion or to be translated in high level programming languages such as Pascal and C for compilation. A simple example is presented next to illustrate the foregoing concepts.

Example: Single-Server Queueing Model

An event graph for a single-server queueing system is developed. Suppose that customers arrive into the system every ta time units and it takes ts time units to serve each customer. The state variables used in this model are defined as follows:

- Q represents the number of customers waiting for service, and
- S denotes the status of the server with 0 ≡ busy and 1 ≡ idle.

The edge conditions for the model are:

(i) (The server is idle): S = 1,

(ii) (Customers are waiting to be served): Q > 0.

The event descriptions are presented in Table 1 while the event graph is depicted in Figure 1.

Several variations of event graphs have appeared in the literature. Pegden [1985], Hoover and Perry [1989], and Law and Kelton [1991] use these networks to construct event-scheduling discrete event simulations. Schruben [1991] presents SIGMA, a microcomputer implementation of event graphs aimed at teaching the principles of discrete event simulation.

Table 1: Event Descriptions

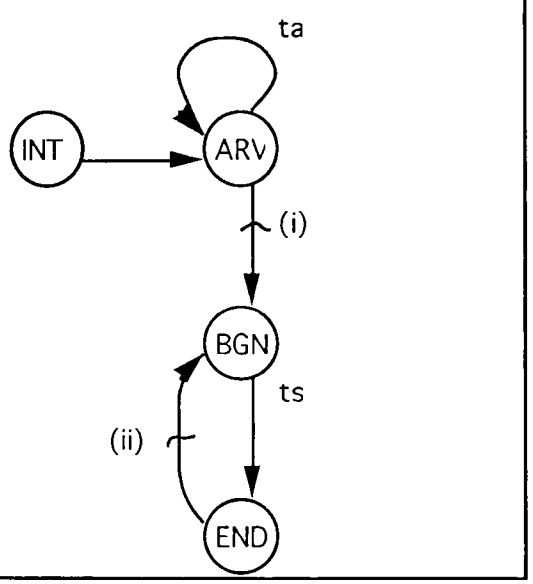
Type	Description	State Changes
INT	Initialization	Q ← 0 S ← 1
ARV	Customer Arrival	Q ← Q + 1
BGN	Begin Service	S ← 0 Q ← Q - 1
END	End of Service	S ← 1

Extensions of event graph analysis have also been introduced. Sargent [1988] presents new rules for detection of simultaneously scheduled events and for event reduction. He also models a flexible manufacturing system with event graphs. In a more recent paper, Som and Sargent [1989] define rules for identifying simultaneously scheduled events and assigning execution priorities. They also introduce a conceptual algorithm for event reduction.

Simulation Graphs [Yücesan and Schruben,

1993] represent an extension in another dimension. The references cited above mainly focus on issues related to the implementation of an event-scheduling simulation model. Schruben and Yücesan, on the other hand, focus mainly on model specification. Their objective is to characterize these networks as a formalism for developing discrete-event models. Moreover, their treatment is not limited to the event-scheduling world view, but targets discrete event dynamic systems in general. For example, a transformation procedure, namely the geometric dual, is shown to yield equivalent representations of queueing models under both event-scheduling and activity scanning world views [Schruben and Yücesan, 1989]. It is indeed this change of focus that lead them to rename these networks *Simulation Graphs*.

Fig.1. Single-Server Queue



Graph theory is used as the foundation of this formalism. A directed graph is characterized as an ordered triple (V(G), E(G), Ψ_G) consisting of a nonempty set of vertices, V(G), a set E(G), disjoint from V(G), of edges, and an incidence function, Ψ_G, that associates with each edge of G a pair of (not necessarily) distinct vertices of G [Bondy and Murty, 1976]. A network is then defined as a graph in which additional data are stored in the vertices and edges [Lawler, 1976]. A *Simulation Graph* is then defined as an ordered quadruple,

$$G = (V(G), E_S(G), E_C(G), \Psi_G),$$

where V(G) is the set of event vertices, $E_S(G)$ is the set of scheduling edges, $E_C(G)$ is the set of cancelling edges, and Ψ_G is the incidence function. The entities in this network are then defined as the following ordered sets:

$\mathcal{F} = \{f_v : v \in V(G)\}$, the set of *state transition functions* associated with vertex v,

$\mathcal{C} = \{C_e : e \in E_S(G) \cup E_C(G)\}$, the set of *edge conditions*,

$\mathcal{T} = \{t_e : e \in E_S(G)\}$, the set of *edge delay times*,

$\Gamma = \{\gamma_e : e \in E_S(G) \cup E_C(G)\}$, the set of *event execution priorities*.

The key idea is that a Simulation Graph specifies the relationships among the elements of the sets of entities in a simulation model. Then, a *Simulation Graph Model* is defined as:

$$\mathcal{S} = (\mathcal{F}, \mathcal{C}, \mathcal{T}, \Gamma, G).$$

The first four sets in the above five-tuple define the entities in a model. The role played by the Simulation Graph, G, in the definition of a Simulation Graph Model, $\mathcal{S}$, is analogous to the role of the incidence function, Ψ, in the definition of a directed graph: it organizes these sets of entities into a meaningful simulation model specification. That is, G specifies the relationships between the elements in the sets $\mathcal{F}$, $\mathcal{C}$, $\mathcal{T}$, and Γ.

Example (Continued):

The entities in the network depicting a single-server queueing system are defined as follows:

$\mathcal{F} = \{f_{INT}, f_{ARV}, f_{BGN}, f_{END}\}$ = { S←1, Q←0; Q←Q+1; S←0, Q←Q-1; S←1}.

$\mathcal{C} = \{C_{ARV,BGN}, C_{END,BGN}\} = \{S=1, Q>0\}$.

$\mathcal{T} = \{t_{ARV,ARV}, t_{BGN,END}\} = \{t_a, t_s\}$.

Γ: to be assigned to ensure the correct execution of the model.

G: see Figure 1.

3 MODEL COMPLEXITY: SOME INTUITIVE MEASURES

The most intuitive measure to predict model complexity is probably the physical size of the program that implements the model specification. In fact, Overstreet [1982] reports that there is a strong relationship between program size and development costs. Hence, a useful complexity measure for Simulation Graph Models is

$$C_1 = |V(G)|$$

which denotes the *cardinality* of the vertex set of a Simulation Graph, G. Note that this metric ignores the potential complexity of an event since events are defined to be "simple" or "atomic" in this paper as they reflect potential modifications in the values of state variables.

In addition, the physical size could be a misleading criterion for judging the overall complexity of a simulation model. Interaction among different modules within the model can be a significant contributor to model complexity. Therefore, a more realistic measure of complexity should also incorporate the impact of *potential intermodular interaction*. One such measure is the *edge-to-vertex ratio*, which is defined as:

$$C_2 = |E(G)| / |V(G)|$$

where $E(G) = E_S(G) \cup E_C(G)$. Simulation Graphs are connected directed graphs. Hence, the minimum edge-to-vertex ratio is (n-1)/n, where n is the cardinality of the vertex set. As Overstreet [1982] points out, *"this represents the simplest possible case, one with no branching. Any execution must consist of a fixed*

sequence of actions, each of which occurs exactly once. The more potential branching after each action, the more complex the behavior of the model." This assertion is consistent with McCabe's [1976] observation that complexity depends only on the decision structure of a program while the addition or deletion of functional statements leaves it unchanged.

A more realistic measure of complexity may be defined as the sum of each module's complexity and the intermodular interaction. In the context of Simulation Graphs, if we let C(v) denote the complexity of vertex v and I(v,w) denote some measure of interaction between (not necessarily distinct) vertices v and w, then the complexity of the model is given by

$$C_3 = \sum_{v \in V(G)} C(v) + \sum_{v,w \in V(G)} I(v,w)$$

Even if the form of the above measure is acceptable, there are no clear choices for C(v) and I(v,w). One approach is to let C(v) represent the number of distinct state changes associated with vertex v. Alternatively, C(v) can be defined as the number of distinct state variables associated with the state transition function, f_v. We will also let I(v,w) represent the number of edges directed from vertex v to vertex w as a measure of interaction. In other words, the second term in C_3 is simply equal to the cardinality of the edge set of the Simulation Graph.

The intuitive measures of complexity discussed thus far can be generally referred to as *space complexity*. The latter is defined as the space needed by an algorithm (in our case, model description) as a function of the size of the problem [Aho et al., 1974]. One can then associate different cost criteria to quantify model complexity. For example, a *uniform cost criterion* would assume that each entity requires one unit of space whereas a *logarithmic cost criterion* would assume that the cost of storing each entity is proportional to its length.

4 MODEL COMPLEXITY: CYCLOMATIC NUMBER

4.1 Motivation and Preliminary Definitions

The complexity measure discussed here is adapted from [McCabe, 1976] who uses it to determine the complexity of computer programs. As will be shown shortly, the scheme presented by McCabe applies to Simulation Graphs quite naturally with almost no modifications.

The complexity measure reflects the number of *control paths* through a program. Since a program with a backward branch potentially has an infinite number of paths, the measure developed here is defined in terms of the *basic paths*.

Definition: A set of paths is called *basic* if appropriate linear combinations of these paths generate every possible path through the program. In addition, paths are called *linearly independent* if they cannot be expressed as linear combinations of other paths.

The following mathematical preliminaries are needed in subsequent derivations. The details can be found in any book on Graph Theory (see, for instance, [Berge, 1973]).

Definition: The *cyclomatic number* of a graph G with n vertices, e edges, and p connected components is defined as

$$\eta(G) = e - n + p.$$

Result: In a strongly connected graph, the cyclomatic number is equal to the maximum number of linearly independent cycles.

This result has implications beyond the application in determining model complexity. In electrical network theory, for instance, $\eta(G)$, the maximum number of linearly independent cycles, corresponds to the largest number of independent circular currents that can flow in the network.

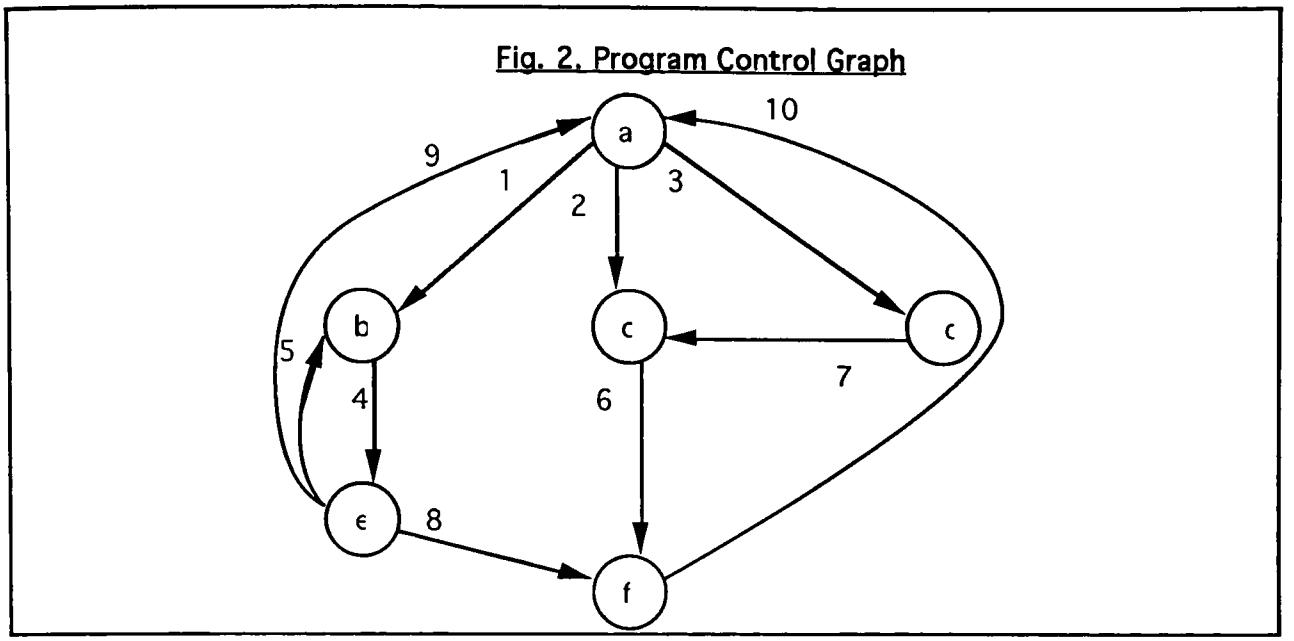
In determining the complexity of a computer program, the above result is used along with a *program control graph*. This directed graph with unique entry and exit vertices is associated with a program in the following manner: each vertex on the graph corresponds to a block of code in the program where control flow is sequential. The edges, on the other hand, correspond to branches taken in the program. It is assumed that each vertex can be reached from the entry vertex, and each vertex can reach the exit vertex.

Suppose that Figure 2 depicts a program control graph with entry vertex a and exit vertex f. To apply the above result, one adds an extra edge from the exit vertex to the entry vertex (edge 10). The control graph then becomes strongly connected and the cyclomatic number is equal to 10-6+1=5, which, in turn, is equal to the maximum number of linearly independent cycles. One can then choose a basis B for the set of all cycles, and any path through the control graph can be expressed as a linear combination of cycles from B.

To illustrate, let B be the following basis:
B: (a b e f a), (b e b), (a b e a), (a c f a), (a d c f a).
To each member of the basis B, we associate a vector as follows:

vector \ edges	1	2	3	4	5	6	7	8	9	10
(a b e f a)	1	0	0	1	0	0	0	1	0	1
(b e b)	0	0	0	1	1	0	0	0	0	0
(a b e a)	1	0	0	1	0	0	0	0	1	0
(a c f a)	0	1	0	0	0	1	0	0	0	1
(a d c f a)	0	0	1	0	0	1	1	0	0	1

Then the path (a b e a b e b e b e f) can be expressed as a vector addition of (a b e a) + 2(b e b) + (a b e f a). In fact, any path through the graph can be expressed as a linear combination of the members of the basis B.



Note that the sequence of an arbitrary number of vertices always has unit cyclomatic complexity. Some properties are listed below:

1. $\eta(G) \geq 1$.
2. $\eta(G)$ is the maximum number of linearly independent paths in G; it is the size of a basis set.
3. Inserting or deleting functional statements to G does not affect $\eta(G)$.
4. G has only one path if and only if $\eta(G)=1$.
5. Inserting a new edge in G increases $\eta(G)$ by unity.
6. $\eta(G)$ depends only on the decision structure of G.
7. $\eta(U_i G_i) = \Sigma_i \eta(G_i)$.

For connected planar graphs, the computation of the cyclomatic number is simplified. Let G be a planar graph with n vertices, e edges, and f faces. Then, by Euler's formula, we have n - e + f = 2. Note that control graphs are strongly connected; then p = 1. That is, $\eta(G) = e - n + 1 = f - 1$. Hence, the complexity of a program with a planar control graph is equal to the number of faces minus one.

4.2 Simulation Graph Complexity

Simulation Graphs are analogous to program control graphs. On a Simulation Graph, each vertex is associated with a set of changes to state variables, which corresponds to a block of code in the program where the flow is sequential. The edges in a Simulation Graph basically determine under what conditions and after how long of a time delay one event will schedule or cancel another one.

Without loss of generality, we can assume that Simulation Graphs have unique entry and exit vertices. An *initialization vertex*, where the initial conditions of the model are established, is the entry vertex. The *exit vertex*, on the other hand, is a special vertex on the Simulation Graph, which terminates the execution of the simulation run upon the satisfaction of the termination conditions.

Simulation Graphs are connected directed graphs. Then, by applying the results of the previous subsection, we define the *cyclomatic complexity* of a Simulation Graph Model as $\eta(G) = e - n + p$, where p = 1. Equivalently, $\eta(G) = f - 1$, where f is the number of faces of G.

It is then possible to measure the complexity of a simulation model by computing the number of linearly independent paths on the associated Simulation Graph and use the cyclomatic number as the basis for testing for model verification and classification for selecting a reasonable test bed of simulations.

5 EXAMPLES

The complexity measures discussed in §§3 and 4 are illustrated on a series of examples. These examples show that the complexity measures, C_1, C_2, C_3, and $\eta(G)$, are indeed reasonable reflections of the requirements imposed by models on computational resources.

All of the tested models are taken from Schruben [1991]. The first model is a single-server queue described in our example and depicted in Figure 1. The second model captures the operations of a semi-automatic machine with breakdowns (ibid, p.60-62). The third model is a job shop (ibid, p.99-107); the fourth represents a flow shop (ibid, p.86); the fifth models a simple assembly process (ibid, p.85). These models are implemented and executed in SIGMA for a single run of 1000 service completions on a personal computer with a 486 processor and 33MHz clock speed. The run times are also depicted in Table 2 together with the proposed complexity measures.

One interesting point to note is that the complexity of individual events, measured as the number of modified state variables within that event, plays a principal role in determining the execution time of a single run, while the branching structure, measured by the cyclomatic complexity of the graph, has secondary influence. This contradicts both Overstreet's [1982] and McCabe's [1976] assertions

that complexity depends only on the decision structure (branching) of a program and not on the number of functional statements.

Table 2: Complexity Measures

Model	C1	C2	C3	η(G)	R.Time
I	4	5/4	11	3	12.4 sec
II	7	11/7	19	6	23.4 sec
III	9	12/9	49	5	256 sec
IV	5	10/5	23	7	34.4 sec
V	6	10/6	31	6	65.4 sec

6 RUN TIME COMPLEXITY

During the execution of simulation models, a considerable portion of the execution time is spent on maintaining the future events list. Comfort [1981] reports that approximately 40% of all the instructions executed during a simulation run is devoted to managing the events list. Henriksen [1983] asserts that, in models of telecommunication systems, total run times can easily differ by as much as 5:1 depending on the choice of the events list algorithm. Special attention has been given to designing efficient list management techniques (see, for example, [McCormack and Sargent, 1981]).

Fox [1987c] presents an efficient alternative for systems that can be modeled as a continuous-time Markov chain. In this setting, instead of using an events list, he improves the execution speed through direct generation of the state transitions by producing rows of the transition matrix as they are needed.

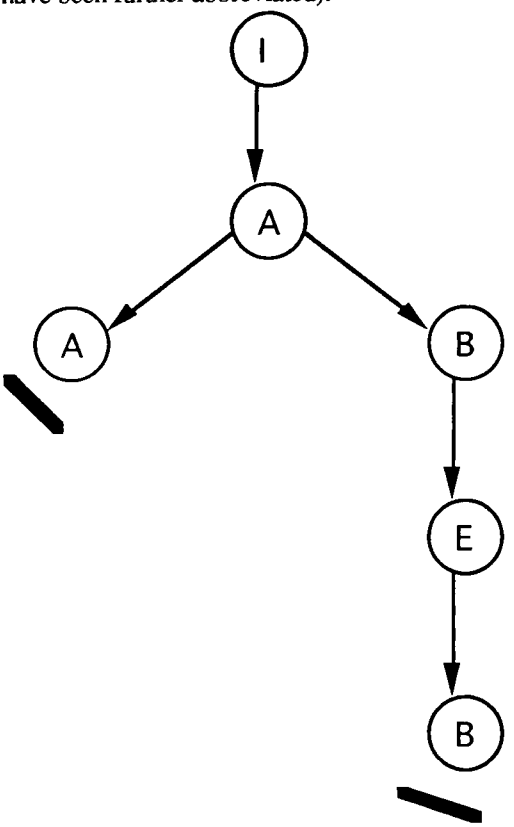
It is therefore natural to use the size of the events list maintained during the execution of a simulation model as a surrogate measure for the run time complexity of that model. In this section, we present a technique based on the Simulation Graphs for determining both the maximum and minimum size of the events list. The technique is based on constructing an event tree and examining its leaves. Our objective is different from previous events list analysis procedures such as the SIMULA HOLD model (see [Devroye, 1986; p.737] for a description). We are simply trying to establish upper and lower limits on the size of the events list. This type of information can be used to assess the memory requirements for executing a simulation. For example, appropriate dimensionality can be determined for NSET and QSET as well as the LIMITS statement in SLAM II [Pritsker, 1986] using this information. We will describe the method in more detail next.

We are assuming again, without loss of generality, that the Simulation Graph, G, has an initialization vertex (entry vertex of Section 4.1), I, where the initial conditions of the model are established. We are also assuming that no vertex or edge attributes are used in the model.

The *event tree* (ET) is then constructed as follows: The root of the tree is the initialization vertex, I. Next, suppose we are at vertex X. Then, for all $Y \in V(G)$ such that $(X, Y) \in E_S(G)$, add vertex Y to the tree along with a directed edge from X to Y. If (X, Y) is a conditional edge on the Simulation Graph, then denote this condition on the event tree as well.

Example (continued):

The Event Tree associated with the Simulation Graph of Figure 1, which is a single-server queueing model, is depicted below (where vertex labels have been further abbreviated):



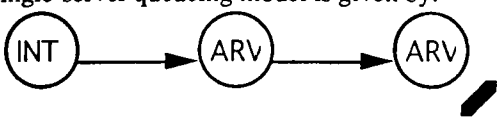
As illustrated by this small example, an Event Tree is a directed tree rooted at the initialization vertex. The vertices on the tree represent events of the Simulation Graph Model, and edges represent how one event schedules further events. Moreover, the possible explosion of the tree is avoided through the application of the following *Fathoming Rule: During the construction of the Event Tree, if an existing vertex is reproduced on the path from the root, the duplicate vertex becomes a terminal vertex. Since the new vertex is identical to its previous copy, all of the events that can be scheduled from it have already been added to the tree by the earlier identical vertex.*

Also note that, since our major interest is in determining the maximum possible size of an events list, cancelling edges are ignored during the construction of the Event Tree. This is because cancelling edges may reduce the size of the events list by deleting some of the event notices, rather than increasing its size by inserting new ones into the events list as scheduling edges do.

We also define an *Unconditional Event Tree* (UET) which contains only those events that are scheduled unconditionally. UET is constructed as follows: The root of the tree is the initialization vertex, I. Next, suppose we are at vertex X. Then, for all $Y \in V(G)$ such that (i) $(X,Y) \in E_S(G)$ and (ii) $C_{X,Y} = \emptyset$, add vertex Y to the tree along with a directed edge from X to Y.

Example (continued):

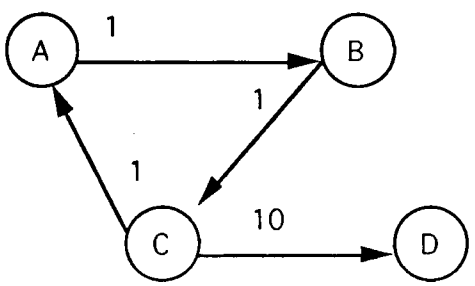
The Unconditional Event Tree associated with the single-server queueing model is given by:



Note that UET can also be obtained from ET by starting at the root of ET and simply deleting all conditional edges along with the subtree onto which the conditional edge is incident. Also note that the same fathoming rule still applies. We next present our surrogate measure of run time complexity.

Lemma: In a Simulation Graph, if there exists a directed cycle with the number of out edges greater than or equal to the number of in edges, then the events list can grow without bound during the execution of the model, provided that the sum of the edge delay times around the directed cycle is strictly less than the delay time on any of the outgoing edges.

Proof: Consider the following directed cycle in a Simulation Graph:



A hand trace through time of the contents of the events list starting with the execution of event A shows that the events list will grow infinitely large. □

Proposition 1: In the absence of the conditions cited in the above lemma, the maximum size of the events list at any instant during the execution of a Simulation Graph Model is equal to the number of leaves of the associated Event Tree.

Proposition 2: In the absence of the conditions cited in the above lemma, the minimum size of the events list at any instant during the execution of a Simulation Graph Model is equal to the number of leaves of the associated Unconditional Event Tree.

Proofs of these propositions are presented in Yücesan [1989]. The interval defined by the minimum and the maximum size of the events list will be referred to as the *complexity interval* of a Simulation Graph Model. Note that all of the models considered in §5 satisfy the conditions of the above propositions. Therefore, the events lists will not grow without bound during the execution of their computer implementations.

The complexity interval can be used as a guide in selecting one of a group of equivalent simulation models. Under the reasonable assumption that smaller events lists lead to faster model executions, the model whose complexity interval is uniformly dominated by that of all other equivalent models should be selected for implementation on a computer.

7 CONCLUDING COMMENTS

Complexity of a simulation model is defined as a measure that reflects the requirements imposed by models on computational resources. All of the measures introduced in this paper are related to the structural properties of models, emphasizing mainly implementation and maintenance efforts. Only one surrogate measure for run time complexity is presented. This is because computational complexity is meaningful only within the context of a specific problem. For example, Fox [1987a,b; 1988a,b] studies finite-horizon, continuous-time Markov chains and compares computational complexities of several methods, including simulation, to estimate such quantities as expected terminal reward, expected cumulative reward, hitting time distribution, and expected reward up to absorption. His complexity criterion is the order of magnitude of the work required to satisfy a given root-mean-square-error tolerance, while assuming that each arithmetic operation and comparison is done without round-off error in O(1) time. The overall simulation effort is then the total number of arithmetic operations and comparisons over all runs. Similarly, computational complexities of algorithms for estimating gradients via simulation are also discussed in Fox and Glynn [1988], Heidelberger et al [1988], and Reiman and Weiss [1989].

One should also note that computational complexity is not the sole, or necessarily even the most important, criterion for evaluating algorithms [Aho et al., 1983]. A complicated but efficient algorithm may not always be desirable because a person other than the writer may have to maintain the program later. In simulation modeling, therefore, a trade-off should always be made between a model that is easy to understand, code and debug, and one that makes efficient use of the available resources. Moreover, this assessment should be a dynamic process with different priorities at different stages of a simulation study. For instance, during the verification phase, a model that is easy to code and debug is desirable whereas, during the experimentation phase, a fast implementation is preferred.

We emphasize once again that measures of complexity for simulation models would find ready utility in supporting the work of both practitioners and researchers. For instance, a reasonable measure of complexity is useful in *a priori* evaluation of proposed simulation studies that must be completed within a specified budget. It can also be useful in classifying simulation models in order to obtain a thorough test bed of models to be used in simulation methodology research.

REFERENCES

Aho, A.V., J.E. Hopcroft, and J.D. Ullman (1974)

The Design and Analysis of Computer Algorithms. Addison-Wesley. Reading, MA.

Aho, A.V., J.E. Hopcroft, and J.D. Ullman (1983) *Data Structures and Algorithms.* Addison-Wesley. Reading, MA.

Berge, C. (1973) *Graphs and Hypergraphs.* North-Holland. Amsterdam, The Netherlands

Bondy, J.A. and U.S.R. Murty (1976) *Graph Theory with Applications.* North-Holland. New York, NY.

CACI - Consolidated Analysis Centers, Inc. (1976) *Simscript II.5 Reference Handbook.* Los Angeles, CA.

Comfort, J.C. (1981) The Simulation of a Microprocessor-Based Event Set Processor. Proceedings of the Fourteenth Annual Simulation Symposium. Tampa, FL. 17-21

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag. New York, NY.

Evans, G.W., G.F. Wallace, and G.L. Sutherland (1967) *Simulation Using Digital Computers.* Prentice Hall. Englewood Cliffs, NJ.

Fox, B.L. (1987a) Gradient Computation for Transient Markov Chains. Technical Report #747. School of OR&IE. Cornell University. Ithaca, NY.

Fox, B.L. (1978b) Generating Markov Chain Transitions Efficiently. Technical Report #758. School of OR&IE. Cornell University. Ithaca, NY.

Fox, B.L. (1987c) Beating Future-Event Schedules. Technical Report #761School of OR&IE. Cornell University. Ithaca, NY.

Fox, B.L. (1988a) Numerical Methods for Transient Markov Chains. Technical Report #810. School of OR&IE. Cornell University. Ithaca, NY.

Fox, B.L. (1988b) Complexity of Gradient Estimation for Transient Markov Chains. Technical Report #753. School of OR&IE. Cornell University. Ithaca, NY.

Fox, B.L. and P.W. Glynn (1988) Replication Schemes for Limiting Expectations. Technical Report #778. School of OR&IE. Cornell University. Ithaca, NY.

Heidelberger, P., X.R. Cao, M.A. Zazanis, and R. Suri (1988) Convergence Properties of Infinitesimal Perturbation Analysis Estimates. *Management Science.* Vol.34.11, 1281-1302

Henriksen, J.O. (1983) Event List Management - A Tutorial. *Proceedings of the Winter Simulation Conference* (Roberts, Banks, and Schmeiser, eds.), 543-552

Hoover, S.V. and R.F. Perry (1989) *Simulation: A Problem Solving Approach.* Addison-Wesley. Reading, MA.

Law, A.M. and W.D. Kelton (1991) *Simulation Modeling and Analysis.* 2nd Edition. McGraw Hill. New York, NY.

Lawler, E. (1976) *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston. New York, NY.

McCabe, T.J. (1976) A Complexity Measure. *IEEE Transactions on Software Engineering.* Vol. SE-2.4, 308-320

McCormack, W.M. and R.G. Sargent (1981) Analysis of Future Event Set Algorithms for Discrete Event Simulation. *Comm. ACM.* Vol.24.12, 801-812

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

Pegden, C.D. (1985) *Introduction to SIMAN.* Systems Modeling Corp. State College, PA.

Pritsker, A.A.B. (1986) Introduction to Simulation and SLAM II. 3rd Edition. John Wiley & Sons. New York, NY.

Reiman, M.I. and A. Weiss (1989) Sensitivity Analysis for Simulations via Likelihood Ratios. *Operations Research.* Vol.37.5, 830-844

Sargent, R.G. (1988) Event Graph Modeling for Simulation with an Application to Flexible Manufacturing Systems. *Management Science.* Vol.34.10, 1231-1251

Schruben, L. (1983) Simulation Modeling with Event Graphs. *Comm. ACM.* Vol.26.11, 957-963

Schruben, L. (1991) *Sigma: A Graphical Simulation System.* The Scientific Press. San Fransisco, CA.

Schruben, L. and E. Yücesan (1987) On the Generality of Simulation Graphs. Technical Report #773. School of OR&IE. Cornell University. Ithaca, NY.

Schruben, L. and E. Yücesan (1989) Simulation Graph Duality: A World View Transformation for Simple Queueing Models. *Proceedings of the Winter Simulation Conference* (McNair, Musselman, and Heidelberger, eds.), 738-745

Som, T.K. and R.G. Sargent (1989) A Formal

Development of Event Graphs as an Aid to Structured and Efficient Simulation Programs. *ORSA Journal on Computing*. Vol.1.2, 107-125

Yücesan, E. (1989) Simulation Graphs for Design and Analysis of Discrete Event Simulation Models. Unpublished PhD Dissertation. School of OR&IE, Cornell University. Ithaca, NY.

Yücesan, E. (1990) Analysis of Markov Chains Using Simulation Graph Models. *Proceedings of the Winter Simulation Conference* (Balci, Sadowski, and Nance, eds.), 468-471

Yücesan E. and L. Schruben (1993) "Modeling Paradigms for Discrete Event Simulation." Forthcoming in *Operations Research Letters.*

Zeigler, B.P. (1976) *Theory of Modeling and Simulation.* John Wiley. New York, NY.

AUTHOR BIOGRAPHIES

LEE SCHRUBEN is a Professor in the School of Operations Research and Industrial Engineering at Cornell University. His research interests are in the design and analysis of large scale simulation experiments. He is a principal developer of SIGMA simulation system. Three of his papers have received outstanding publications awards from the TIMS College on Simulation and the Chemical Division of ASCQ.

ENVER YUCESAN is an Associate Professor of Operations Research at the European Institute of Business Administration (INSEAD) in Fontainebleau, France. He has a BSIE degree from Purdue University, and MS and PhD degrees in OR from Cornell University.