REVISED DRAFT

27 June 1984

Technical Report CS83028-R

GRAPH-BASED DIAGNOSIS OF DISCRETE EVENT MODEL SPECIFICATIONS*

C. Michael Overstreet+
and
Richard E. Nance++

---

*Research partially supported by the Office of Naval Research and the Naval Sea Systems Command through the System Research Center at Virginia Tech.
+Department of Computer Science, Old Dominion University, Norfolk, Virginia 23508.
++Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061.

Abstract

Several diagnostics which assist in the construction of specifications for discrete event simulation models are defined. The model specifications to be analyzed must be in a particular form called a Condition Specification. The diagnostics are based on analysis of graphs easily derived from a Condition Specification. Most of the diagnostics are intended to be applied as a Condition Specification is being developed. Three categories for the classification of diagnostics of model specifications are defined. Two examples illustrate the graphical forms derivable from a Condition Specification.

CR Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications -- languages, methodologies; D.2.5 [Software Engineering]: Testing and Debugging -- diagnostics; I.6.m [Simulation and Modeling]: Miscellaneous -- model specification and diagnosis.

General Terms: Design, Documentation, Languages

Additional Key Words and Phrases: discrete event simulation, model specification languages, model specification diagnosis, graph based diagnosis

1. Introduction

Software development technology is recognized as an area in which major advances are needed. Various systems and methodologies have been advocated for reducing software costs (see, for example, [HOWDW82, WASSA81]). The work reported here focuses on *model development*, which encompasses more than "program development" [NANCR83, p. 326]. By restricting attention to the modeling domain of discrete event simulation, we hope to identify techniques which can reduce cost and improve the quality of simulation models.

Several researchers have reported work in the area of simulation support, perhaps the most interesting being the program generators [CLEMA78, DAVIN76, MATHS75, VIDAC80]. Other authorities have asserted that, if significant improvements are to be realized, the problems of quality and productivity must be addressed at a global level through the creation of integrated systems designed to provide an environment for simulation model development and experimentation [NANCR81b, ORENT82, ZEIGB84]. In this paper we discuss some tools which are intended to be part of a Model Development Environment (MDE) defined by Balci [BALCO83].

While different tools are required in different phases of the model life cycle, we concentrate on diagnostic procedures intended to support the model development phase of the process. These tools assist the development process by providing early diagnosis of several types of real or potential problems and by providing several forms of documentation.

2. The Role of Diagnosis in a MDE

Work is underway at Virginia Tech to define a MDE. The MDE definition is to be based on experience gained through experimentation with several prototypes. The prototypes assume the existence of a sufficiently robust constellation of hardware, languages, data bases, operating systems, and technically competent people to provide an effective modeling environment. See [BALC083] for a discussion of the toolset currently comprising this environment.

2.1 The Underlying Methodology

The prototype MDEs are designed to reflect an underlying approach to model building called the Conical Methodology [NANCR81a]. The Conical Methodology requires, among other things, that the modeling team employ a top-down model definition followed by bottom-up model specification that realizes model documentation during the construction process. Additionally, project documentation is an indispensable requirement of the MDE. Important in the Conical Methodology is the clear distinction between a "model specification" (which describes how the model is to behave) and a "model implementation" (which describes how the behavior is to be achieved).

2.2 Model Diagnosis

The purposes of model diagnosis are:

(1) to assist in the identification of conceptual errors (misperceptions) or descriptive errors (misrepresentations) as early as possible in the modeling effort,

(2) to suggest alternatives that might be less prone to errors or might offer more efficient model development and experimentation, and

(3) to provide guidance and checks on the modeling effort.

Diagnosis begins with the first communicative model [NANCR81a, p. 14] and continues throughout the model life cycle [NANC81a, pp. 12-17].

Three categories of diagnostic assistance are identified within the MDE. The most common is *analytical diagnosis*: determination of the existence of a property such as attribute utilization or revision consistency (both discussed below). Another category is *comparative diagnosis*, which requires the definition of measures intended to depict differences among multiple representations of a single model or between representations of different models. *Informative diagnosis* forms the third category, which includes assistance in the form of characteristics extracted or derived from model representations. Graphical techniques are prominent in this last category.

The application of graphical techniques presented below require that a model specification be in a particular form. This form, a Condition Specification (CS) [OVERC83], reveals dynamic relationships among the components of a model specification. The CS may be inconvenient for the modeling team to deal with directly. Therefore, a critical component of the MDE is a "model generator," which is utilized by the modeling team in the construction of a CS. By invoking the diagnosis of incomplete model specifications, model verification can be introduced much earlier than is currently attempted or perceived to be possible.

3. Condition Specifications

A Condition Specification (CS) representation treats a model as a collection of objects, each object composed of attributes and perhaps additional objects. The model itself is such an object. The attributes convey sufficient information about the state of each object of properly sequence model actions and record model behavior. The attribute structure of each object in the model is described by a collection of object descriptions.

The dynamic behavior of the model is defined by a collection of Action Clusters (ACs), a descriptive structure perhaps most easily presented through an example: a simple M/M/1 queueing system. The CS language used in this example is described in [OVERC83]. This M/M/1 example also facilitates description of the diagnostics which can be generated from a CS representation.

3.1 Condition Specification Example

The purpose of this example is to illustrate an approach yielding a CS. The example is very simple; a more interesting example is presented in Section 5. The specification presented here consists of the following components:

- a *model abstract*: a concise description of the model in English.
- a *model objective*: a description of the data to be produced by the model.
- an *object structure*: a decomposition of the model into objects and attributes.
- a set of *action clusters*: a specification of the behavior of the model (see below).

Analysis techniques treated herein are restricted to the action clusters. The other components are provided in conformance with the Conical Methodology.

Model Abstract:

> Customers arrive randomly, wait in line if necessary for a single server until the server becomes available. The server attends a customer for a random amount of time, and then assists the next customer in line, if any. If no customers are waiting, the server waits for the next arrival. After service is complete, each customer leaves the system.

Model Objective:

> Estimate the utilization of the server.

The remainder of the Condition Specification for this model is presented in Tables 1 and 2. The object structure of the model is presented in Table 1. The model, "M/M/1 System," consists of the attribute, "system_time," and two objects, "Customer" and "Server." The object Customer consists of two attributes, one a real number and the other a "time-based signal" (explained below). The object Server consists of seven attributes: five which are numeric, another a "time-based signal," and an ordinal type (similar to Pascal's user defined types).

Object: M/M/1 Queueing System	
Attributes:	Attribute Type:
system_time	positive real
Objects:	
Customer	
Attributes:	Attribute Type:
arrival_mean	positive real
arrival	time-based signal
Objects: none	
Server	
Attributes:	Attribute Type:
server_utilization	positive real
queue_size	nonnegative integer
server_status	{ idle, busy }
end_of_service	time-based signal
num_served	nonnegative integer
service_mean	positive real
total busy_time	positive real
Objects: none	

Table 1: M/M/1 Object Structure

A "time-based signal" is used to schedule actions which are to occur at a particular time. This concept is illustrated in Table 2. For example, each arrival schedules a future arrival and each begin service (the third action cluster) schedules an end of service. Each time-based signal is treated as a Boolean variable that is false except for those particular instants of time for which the signal has been scheduled.

The ACs for the M/M/1 System are presented in Table 2. In the CS approach, two special ACs are required; one to define the actions which initiate execution of a model, and the second to describe the conditions leading to termination and the consequent actions that are to take place on termination (typically, some type of run report is generated). These two ACs are included in Table 2. The remaining three ACs describe what happens when:

(1) an arrival occurs: queue size is incremented and the next arrival is scheduled;

(2) service begins: queue size is decremented, the current time is recorded, the server status is changed, and an end of service is scheduled.

(3) the server finishes with a customer: server status is changed, the number of customers served is incremented, and a running total of server busy time is incremented; and

The ACs also describe the causes of each set of actions. Arrival and end of service occur due to scheduling. Beginnings of service and the end of a simulation run occur due to attributes taking on particular values.

Since this example is so simple, several languages constructs generally useful for specification of discrete event simulation models are not included. See [OVERC83] for a description of the language constructs available for CSs.

```
WHEN initialization                    /* Initialization */
    CREATE ( server )
    CREATE ( customer )
    READ ( arrival_mean, service_mean )
    queue_size := 0
    server_status := idle
    num_served := 0
    total_busy_time := 0
    SCHEDULE ( arrival, 0 )
    END WHEN

WHEN arrival                           /* Arrival */
    queue_size := queue_size + 1
    SCHEDULE ( arrival, negexp ( arrival_mean ) )
    END WHEN

WHEN ( queue_size > 0 ) and ( server_status = idle )
    queue_size := queue_size - 1  /* Begin service */
    begin_time := system_time
    server_status := busy
    SCHEDULE ( end_of_service, negexp ( service_mean ) )
    END WHEN

WHEN end_of_service                    /* End service */
    server_status := idle
    total_busy_time := total_busy_time +
      ( system_time - begin_time )
    num_served := num_served + 1
    END WHEN

WHEN num_served > 500                  /* Termination */
    server_utilization := total_busy_time / system_time
    WRITE ( 'Server utilization: ', server_utilization )
    STOP
    END WHEN
```

Table 2: M/M/1 Action Clusters

3.2 Analysis Using the Condition Specification

The analysis techniques to follow are all based on the ability to identify the "causes" for each set of model actions (in terms of the attributes in the condition expressions) and the "effects" of each set of actions (in terms of attributes whose values are altered). The model specification can be

regarded as a set of Action Clusters, each AC in the set consisting of a condition and a set of actions that are to occur whenever that condition is true. This abstract view is illustrated in Figure 1. The analysis routines then examine the interaction between the actions of one AC and the condition expressions of all ACs.

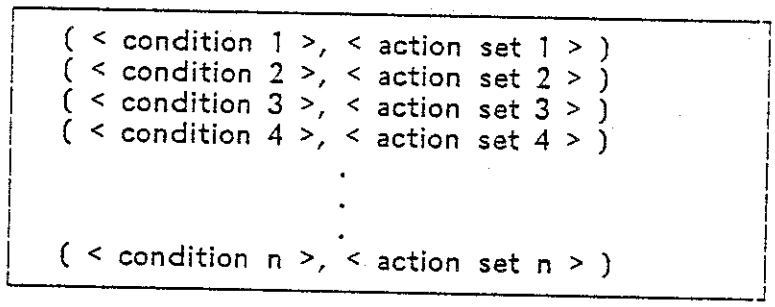


Figure 1: Abstract of Condition Specification Representation

Figure 1 illustrates the strong similarity of the CS representation to production rule organization in knowledge bases [NEWEA72, p. 23-33]. Achieving this resemblance was not an underlying motivation; however, the implications of the structural similarities are not being ignored. Consequently, the utility of logic programming within the Model Generator forms a currently active area of investigation.

Analyses can be applied to partial model specifications, although a more complete specification enables more thorough analysis. Analysis of partial model specifications requires the modeling team to identify both the model attributes which influence a set of model actions and the results of those actions (also in terms of model attributes). Indeed, one of the functions of the "model generator" mentioned above is to elicit this information from the modeling team as the specification is developed. And as a specification

becomes more complete, the consistency of each previously incomplete component can be compared with the more complete version with respect to these attributes.

4. Graph-Based Analysis

Basing analysis of model specifications on a graphical representation of a model specification is not entirely new. DeCarvalho and Crookes [DECAR76] analyze models to identify independent "cells" in order to improve the efficiency of an activity scanning time flow mechanism and to identify components whose output can be saved and reused in replications of the simulation runs. Schruben describes an analysis of "event graphs" to simplify a model specification and to identify other aspects of the model such as a minimal set of events that must be initialized at the start of any execution [SCHRL83].

The principal emphasis in this paper is analysis of model specifications. In this section, two graphs are defined which can be generated directly from a CS. Examples of these graphs, based on the M/M/1 example of Section 3, are included. A third graph is described which can be generated during execution of a model. Several analytical and informative diagnostics based on these graphs are described.

4.1 Action Cluster Attribute Graphs

Construction of the graphs requires definition of three terms.

An attribute x is a *control attribute* of an action cluster if x appears in a condition expression of the action cluster.
An attribute x is an *output attribute* of an action cluster if the actions cluster can change the value of attribute x.
An attribute x is an *input attribute* of an action cluster if the value of x affects the output attributes of the action cluster.

As an example, Table 3 lists the control, input and output attributes for the action clusters of the M/M/1 system of Section 3.

Action Cluster	Control Attributes	Input Attributes	Output Attributes
INITIALIZATION	initialization	arrival_mean service_mean	arrival_mean server_mean queue_size server_status num_served total_busy_time arrival
ARRIVAL	arrival	queue_size arrival_mean	queue_size arrival
BEGIN SERVICE	queue_size server_status	queue_size system_time service_mean	queue_size begin_time server_status end_of_service
END SERVICE	end_of_service	total_busy_time system_time begin_time num_served	server_status total_busy_time num_served
END SIMULATION	num_served	total_busy_time system_time	server_utilization

Table 3: M/M/1 Attribute Classification

For a Condition Specification CS, let

$$T = \{t_1, t_2, \ldots, t_k\}$$

be the set of all time-based signals in CS. Let

$$A = \{a_1, a_2, \ldots, a_m\}$$

be the set of all other attributes in CS. Let

$$AC = \{ac_1, ac_2, \ldots, ac_n\}$$

be the set of all action clusters in CS.

The *Action Cluster Attribute Graph* (ACAG) for a CS is defined as follows. Let CS be a Condition Specification with k time-based signal attributes, m other attributes, and n action clusters. Then G, a directed multigraph with k + m + n nodes, is constructed as follows.

G has a directed edge from node i to node j if

(1) node i is a control or input attribute for node j, an AC, or

(2) node j is an output attribute for node i, an AC.

Thus G is bipartite with one set of nodes representing ACs and the other, attributes.

The ACAG depicts the interaction between the ACs and attributes in a model specification, showing both the ability of an AC to change the value of an attribute, and the ability of an attribute to cause or influence the actions of an AC. In order to distinguish interactions that occur instantly from those involving a time delay, edges from an AC to a time-based signal (attribute) are depicted with a dotted line; other edges with a solid line.

While the graph is a potential documentation tool, any complex model is likely to have a graph much too large to be directly helpful to a modeler. The graph, in the form of an appropriate data structure, is primarily useful as input for some of the analyses discussed below.

4.2 Analysis of Action Cluster Attribute Graphs

The ACAG can be used to support several types of model analysis:

(1) Attribute Utilization: if the node representing an attribute has an out-degree of 0, then that attribute cannot influence model behavior.

Note that having an unutilized attribute is not necessarily an error since the attribute may be serving strictly a statistical function, in which case it should appear in some output statement (and this is easily verified).

(2) Attribute Classification: analysis of the graph can identify the function (e.g. control, output, or input) of each attribute in a model specification.

For example, attributes can be classified as "control," if their value change can cause the occurrence of some actions, or as "statistical," if they serve only reporting functions.

(3) Attribute Initialization: if the node representing an attribute has in-degree of 0, then the attribute must be uninitialized.

The model may contain uninitialized attributes other than those with an in-degree of 0. The absence of Attribute initialization is a sufficient but not a necessary condition for the existence of uninitialized variables.

(4) Action Cluster Completeness: if the condition expression for an AC contains attributes which are not time-based signals, then at least one control attribute of the AC must also be an output attribute of the same AC.

To avoid one type of "infinite loop" some action of the AC must eventually make the condition for that AC false.

(5) Revision Consistency: if the model specification is developed in stages in a MDE, then the intended use of attributes as described in an incomplete specification can be checked against the actual usage as the specification becomes more complete.

4.2.1 Matrix Representations

Matrix representations can be constructed from an ACAG which, while having intuitive appeal, have yet to demonstrate actual utility. These ideas are similar to the term-document matrices from information storage and retrieval [SALTG68, p. 50].

The information contained in an ACAG can be represented by a pair of Boolean matrices. Let a CS have m ACs, ac_1, ac_2, ... , ac_m, and n attributes, a_1, a_2, ... , a_n. Then the *attribute action-cluster matrix* for a CS is an n by m Boolean matrix defined as follows:

$$b(i,j) = \begin{cases} 1 & \text{if edge } (a_i, ac_j) \text{ exists in the ACAG} \\ 0 & \text{otherwise.} \end{cases}$$

The *action-cluster attribute matrix* for a CS is an *m* by *n* Boolean matrix defined as follows:

$$b(i,j) = \begin{cases} 1 & \text{if edge } (ac_i, a_j) \text{ exists in the ACAG} \\ 0 & \text{otherwise.} \end{cases}$$

4.2.2 Operations on Matrix Representation

If A_1 is an attribute action-cluster matrix for a CS and A_2 is an action-cluster attribute matrix for the same CS constructed so that the ordering of attributes and ACs corresponds in both matrices, then we can define the *attribute interaction matrix* A as

$$A = A_1 \times A_2.$$

A contains a 1 in position (i,j) if and only if attribute a_i is an input or control attribute for an AC for which a_j is an output attribute. Thus the matrix A indicates the potential for one attribute to directly influence the value of another.

As the term-document matrices, powers of the matrix A can be used to measure the strength of relationships among attributes and perhaps assist the modeler in decomposing the CS by identifying groups of related attributes. The term "attribute cohesion" is given to the measure of this strength of relationships.

If the order of the matrices in the above multiplication is reversed, A is an *action cluster interaction matrix*. It contains a 1 in position (i,j) if and only if ac_i has an output attribute which is an input attribute of ac_j. Thus an action cluster interaction matrix indicates the ability of one AC to directly influence another.

Again, powers of an action cluster interaction matrix can be used to measure strength of relationships between ACs and perhaps guide the modeler in the decomposition of the CS. The term "action cluster cohesion" is used.

4.3 Action Cluster Incidence Graphs

Another graph, the *Action Cluster Incidence Graph* (ACIG), can be used to assist in the analysis of model specifications. The purpose of an ACIG is to depict one type of interaction between ACs. The idea is simple: if an action of AC_i can cause (by changing the value of a control attribute) AC_j to occur, then a directed arc leads form AC_i to AC_j. As before, we use a solid line to indicate that AC_i can cause AC_j to occur in the same instant of simulated time and a dotted line to indicate that AC_i can cause AC_j to occur after a time delay (through a time-based signal).

It is easy to construct an ACIG from a CS that includes an arc from AC_i to AC_j if AC_i might possibly cause AC_j to occur. A procedure to do this is given below. However, this procedure produces a graph which may include many extra edges. That is, it may include an edge from AC_i to AC_j even though AC_i could never cause AC_j to occur. Overstreet [OVERC82, p. 271] shows that no algorithm can exist to produce an ACIG from any model specification which never includes extra edges. Even so, in most cases a reasonable graph is easily constructed. Some simplification techniques are illustrated below and an example of a graph before and after such simplifications is presented in Section 5.

An ACIG for a CS can be constructed as follows:

Let a CS consist of the set of ACs $\{ac_1, ac_2, \ldots, ac_n\}$.

(1) For each $1 \leq i \leq n$, let $node_i$ represent ac_i.

(2) For each ac_i, construct two sets of attributes. Let T_i be the set of control attributes for ac_i which are time-based signals. Let C_i be the set of control attributes for ac_i which are not time-based signals.

(3) For each ac_i, construct the set O_i containing the output attributes for ac_i.

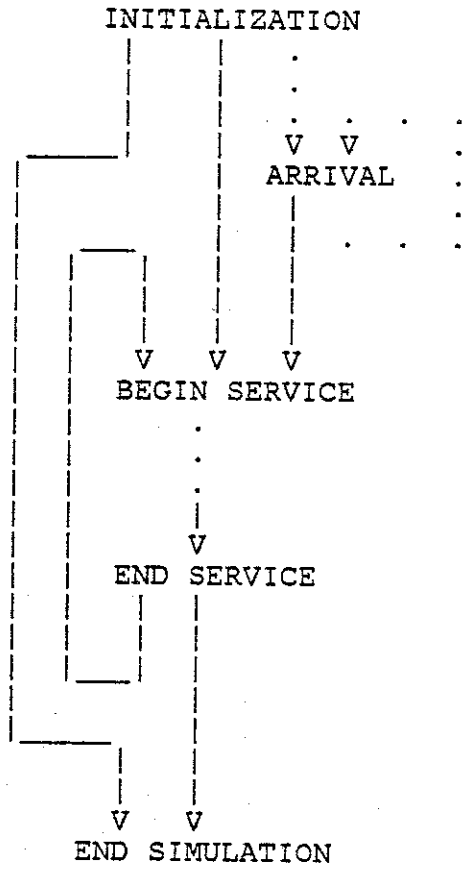
(4) For each $1 \leq i \leq n$,
    For each $1 \leq j \leq n$,
        Draw a solid arc from $node_i$ to $node_j$ if O_i intersect C_j is nonempty.
        Draw a dotted arc form $node_i$ to $node_j$ if O_i intersect T_j is nonempty.
    END for each $1 \leq j \leq n$
END for each $1 \leq i \leq n$.

The ACIG produced by this algorithm for the M/M/1 system is presented in Figure 2. A simple analysis of the CS would allow several simplifications of this graph. For example, the graph of Figure 2 has an arc from the INITIALIZATION AC to the END_SIMULATION AC (since the attribute "num_served," a control attribute for the END_SIMULATION AC, is altered by the INITIALIZATION AC). But the END_SIMULATION only occurs when num_served is greater than 500. INITIALIZATION sets the value of num_served to 0; hence it could not possibly cause the END_SIMULATION to occur in the same instant of time. A simplification technique is discussed below.

4.4 Analysis of Action Cluster Incidence Graphs

The ACIG can assist in several interesting analyses. The graph itself provides the modeling team with documentation on possible interactions among components. Other possibilities include the following:

(1) Connectedness: if, for a given Condition Specification, no Action Cluster Incidence Graph exists which is not connected, then the Condition Specification is said to be connected.



Legend:

--------> Immediate Change
. . . .> Delayed Change

Figure 2: M/M/1 Action Cluster Incidence Graph

If a CS has an ACIG which is not connected, then the model specification contains components which cannot influence the actions of other components. Consequently, the specification consists of two or more independent submodels. The property can be extended in a useful fashion. If the Initialization AC is the only node connecting components, then the specification contains noninteracting components. While the lacks of connectedness is not necessarily an error, knowledge that a CS has this property may be useful to the modeling team.

(2) Accessibility: if, for a given Condition Specification, no Action Cluster Incidence Graph contains nodes with an in-degree of zero (other than the Initialization node) then the Condition Specification is said to be accessible.

If a CS is not accessible, then it contains ACs which can never be activated in any execution based on the model specification. These ACs can be deleted form the model specification without affecting its behavior. For a complete specification, this is probably not what the modeling team intended.

(3) Out-complete: if, for a given Condition Specification, no Action Cluster Incidence Graph contains nodes with an out-degree of zero (other than the termination node), then the Condition Specification is said to be out-complete.

If a CS is not out-complete, then it contains ACs which can be deleted from the model specification without affecting model behavior.

The inclusion of ACs with an out-degree of zero is not necessarily an error. The purpose of such an AC might be to report model behavior. In this case the AC should contain some kind of "write" operation, which is readily recognizable.

(4) Complexity: no definition of complexity is offered here; it is a property not easily defined. However, we propose that an ACIG for a model specification can be analyzed to suggest, at least to a relative if not an absolute degree, both anticipated run-time overhead and difficulty in implementation, verification/validation, and maintenance.

Several graph-based metrics can be proposed which, at least at an intuitive level, relate to the complexity of the underlying model. Several authors have proposed graphical related metrics for the study of software complexity. See, for example, [MCCAT76, WOODM79, HENRS81a] for some proposed metrics and [ELSHJ78, CURTB79, HENRS81b] for evaluations of these metrics.

(5) Precedence structure: An ACIG, by the nature of its construction, conveys something about the sequencing of model actions. For example, if a node has in-degree one, then its occurrence in any execution based on the CS must always be preceded by that particular AC with an arc to that node.

For each AC, the ACIG provides both a set of predecessor ACs and a set of successor ACs. Although no algorithm is likely to detect errors in these sets, the predecessor and successor sets are interesting diagnostic tools. If these lists are presented to an individual who understands the behavior of the system being simulated, that individual may notice errors in the specification if, for example, anticipated successors are missing or unexpected successors are included.

(6) Decomposition: While the ACIG depicts interaction between ACs, the approach can be used for components other than ACs. An incidence graph for decompositions of a model into structures other than ACs and be utilized.

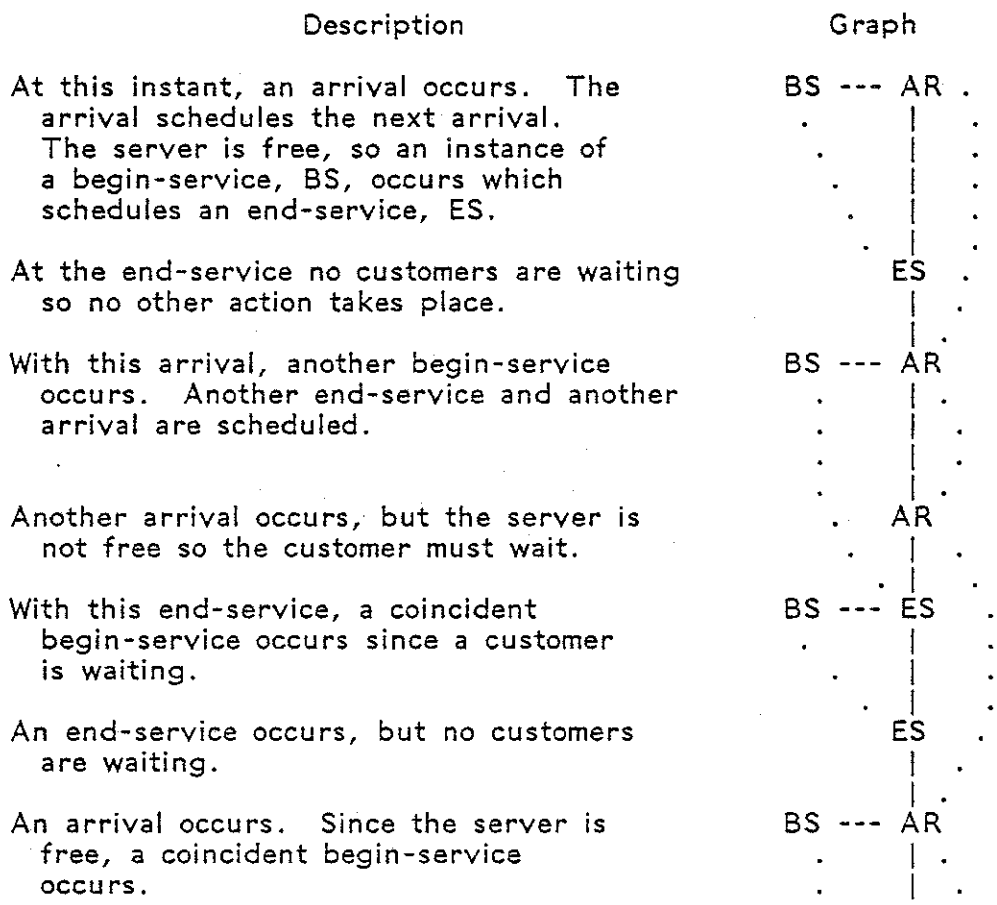
For example in [OVERC82], this approach is used to analyze alternate world view representations of a single CS.

4.5 Run-Time Graphs

The depiction of possible causal relationships among ACs using graphs can contribute to verification in the early stages of model development. As a diagnostic tool, graphs such as the hypothetical run-time graph illustrated in Figure 3 can be generated by a "symbolic execution" of a model specification.

Model Description: Assume a CS contains 3 action clusters. AR, representing an arrival, schedules the next arrival and increments a count of customers awaiting service. ES, an end-service, changes the status of the server to free. BS, a begin-service, has the condition that customers are waiting and the service is free. It schedules an end-service, decrements the count of waiting customers, and changes the status of the server to busy.

Hypothetical Execution:



Then notation in the graph is similar to that of Figure 2. The vertical line represents a time scale.

Figure 3: Run-time Graph Example

An AGIG depicts potential causal relationships in a model specification. The purpose of this graph is to identify actual causal relationships that occur during a simulation run. The notation used in Figure 3 is similar to that used for ACIGs. The vertical line represents a time axis with time increasing down the axis. Thus if AC_1 causes AC_2 to occur in the same instant of time, the ACs are connected by a solid line and are at the same level. If AC_1 schedules the occurrence of AC_2 for a future point in time, the ACs are connected by a dotted line with the scheduled AC lower in the figure than the scheduling AC.

5. An Example -- The Machine Repairman Model

We use a version of the classical machine repairman model as a more complex example to illustrate some of the graphs discussed here. Forms of this model are used in studies by Nance [NANCR71, NANCR81a]. The model is from [PALMD47, COXD61].

Model Abstract:

> A single repairman services a group of *n* identical semiautomatic machines. Each machine requires service randomly based on a time between failure which is a negative exponental random variable with parameter "mean_uptime." The repairman starts in an idle location and, when one or more machines requires service, the repairman travels to the closest machine needing service. Service time for a machine follows a negative exponential distribution with parameter "mean_repairtime." After completing service for a machine, the repairman travels to the closest machine needing service or to the idle location to await the next service request. The closest machine is determined by shortest travel time. Travel time between any two machines or between the idle location and a machine is determined by a function evaluation.

Model Objective:

> Estimate total downtime for each machine.

The ACs for a CS for this model are given a Figure 4, the successors for each AC in Table 4, and the resulting ACIG in Figure 5. For brevity, we have omitted the object structure specification.

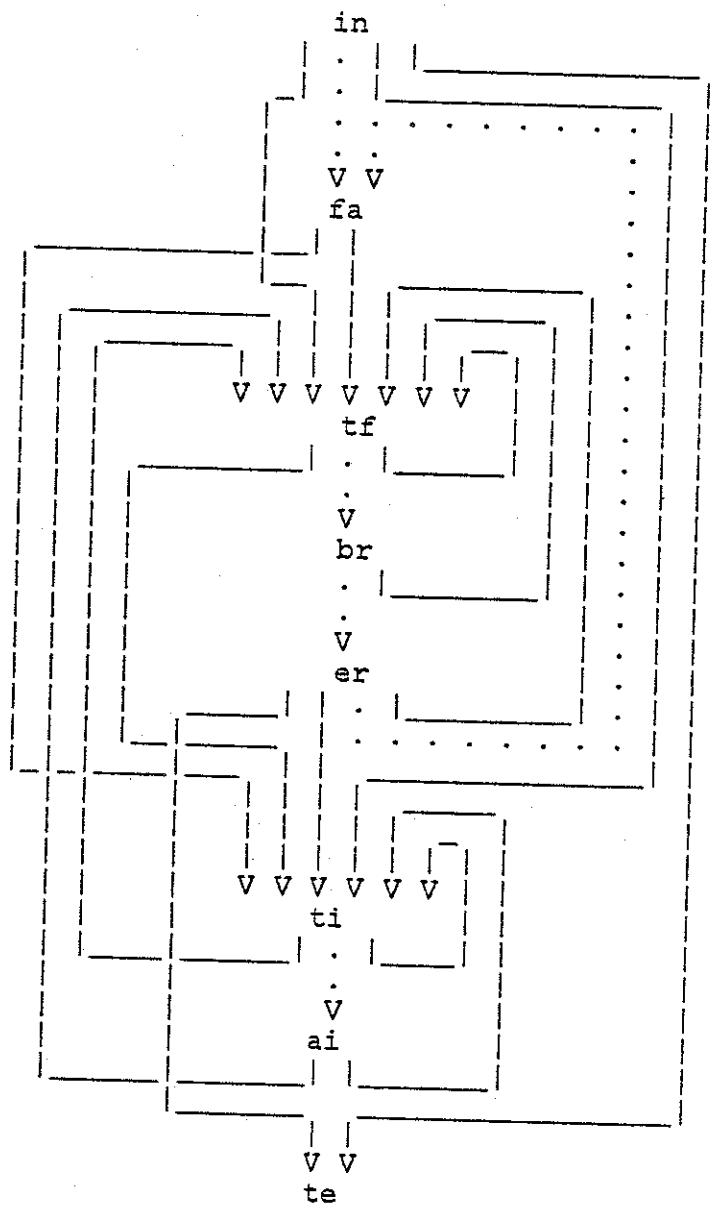
```
WHEN initialization                          /* Initialization */
   READ( n, max_repairs, mean_uptime, mean_repairtime )
   CREATE( repairman )
   FOR i := 1 TO n DO
     CREATE( facility[ i ] )
     fac[ i ] := i
     failed[ i ] := false
     total_down_time[ i ] := Ø
     SCHEDULE(failure( i ), neg_exp( mean_uptime ) )
     END FOR
   num repairs := Ø
   location := idle
   status := avail
   END WHEN

WHEN num_repairs > max_repairs               /* Termination */
   FOR i := 1 TO n DO
      WRITE( 'Total downtime facility ',i,':',total_down_time[ i ] )
      END FOR
   STOP
   END WHEN

WHEN failure( i : 1 .. n )                   /* Failure */
   failed[ i ] := true
   begin_down_time[ i ] := system_time
   END WHEN

WHEN arr_facility( i : 1 .. n )              /* Begin repair */
   SCHEDULE( end_repair( i ), neg_exp( mean_repairtime ) )
   status := busy
   location := fac[ i ]
   END WHEN

WHEN end_repair( i : 1 .. n )                /* End repair */
   SCHEDULE( failure( i ), neg_exp( mean_uptime ) )
   Failed[ i ] := false
   total_down_time[ i ] := total_down_time[ i ] +
      ( system_time - begin_down_time[ i ] )
   status := avail
   num_repairs := num_repairs + 1
   END WHEN

WHEN( FOR ALL 1 <= i <= n,                   /* Travel to idle */
      NOT failed[ i ] ) &
      status = avail &
      location <> idle
   SCHEDULE( arr_idle, traveltime( location, idle ) )
   status := travel
   END WHEN

WHEN arr_idle                                /* Arrive idle */
   status := avail
   location := idle
   END WHEN

WHEN status = avail &                        /* Travel to fac */
      ( FOR SOME 1 <= i <= n,
      failed[ i ] )
   i := closet_failed_fac( failed, location )
   SCHEDULE( arr_facility[ i ], traveltime
      ( location, fac[ i ] ) )
   status := travel
   END WHEN
```

Figure 4: Machine Repairman Action Clusters

Action Cluster	Graph Id	Potential Successors	Successor Type
Initialization	in	tf	c
		ti	c
		te	c
		fa	d
Termination	te	.	.
Failure	fa	ti	c
		tf	c
Begin repair	br	ti	c
		tf	c
		er	d
End repair	er	ti	c
		tf	c
		te	c
		fa	d
Travel to idle	ti	ti	c
		tf	c
		ai	d
Arrive idle	ai	ti	c
		tf	c
Travel to facility	tf	ti	c
		tf	c
		br	c

"." indicates no successors.
"c": contingent action cluster (condition contains no time-based signal).
"d": determined action cluster (condition contains a time-based signal).

Table 4: Machine Repairman Action Cluster Successors



"____" Immediate change.
"...." Time delayed change.

Figure 5: Machine Repairman Action Cluster Incidence Graph

Action Cluster Id	Graph Id	Type (1)	Original Rebuttable Successors(2)	How Pruned (3)	Reduced Rebuttable Successors (2) d	c
Initiali_ zation	in	c	tf: failed(i) status ti: location status failed(i) te: num_repairs max_repairs fa: failure(i)	in => ¬tf: failed(i) has wrong value. in => ¬ti: location has wrong value. in => ¬te: num_repairs has wrong value. No pruning.	.	fa
Termina_ tion	te	c	.	Since termina_ tion action.	.	.
Failure	fa	d	ti: failed(i) tf: failed(i)	fa => ¬ti failed(i) has wrong value. No pruning.	.	tf
Begin repair	br	d	ti: status location tf: status failed(i) er: end_repair	br => ¬ti status has wrong value. br => ¬tf status has wrong value. No pruning.	er	.

(1) "c": contingent action cluster; "d": determined action cluster
(2) "." indicates no successors.
(3) "¬" is a logical negation.

Table 5: Machine Repairman Sucessor Analysis

Action Cluster Id	Graph Id	Type	Original Rebuttable Successors	How Pruned	Reduced Rebuttable Successors d	c
End repair	er	d	ti: failed(i) status	No pruning.	fa	ti
			tf: failed(i) status	No pruning.		tf
			te: num_repairs	No pruning.		te
			fa: failure(i)	No pruning.		
Travel to idle	ti	c	ti: status	ti => ¬ti status has wrong value.	ai	.
			tf: status	ti => ¬tf status has wrong value		
			ai: arr_idle	No pruning.		
Arrive idle	ai	d	ti: status location	ai => ¬ti location has wrong value.	.	tf
			tf: location	No pruning.		
Travel to facili_ ty	tf	c	ti: status	tf => ¬ti status has wrong value.	br	.
			tf: status	tf => ¬tf status has wrong value.		
			br: aff_fac	No pruning.		

Table 5: (Continued)

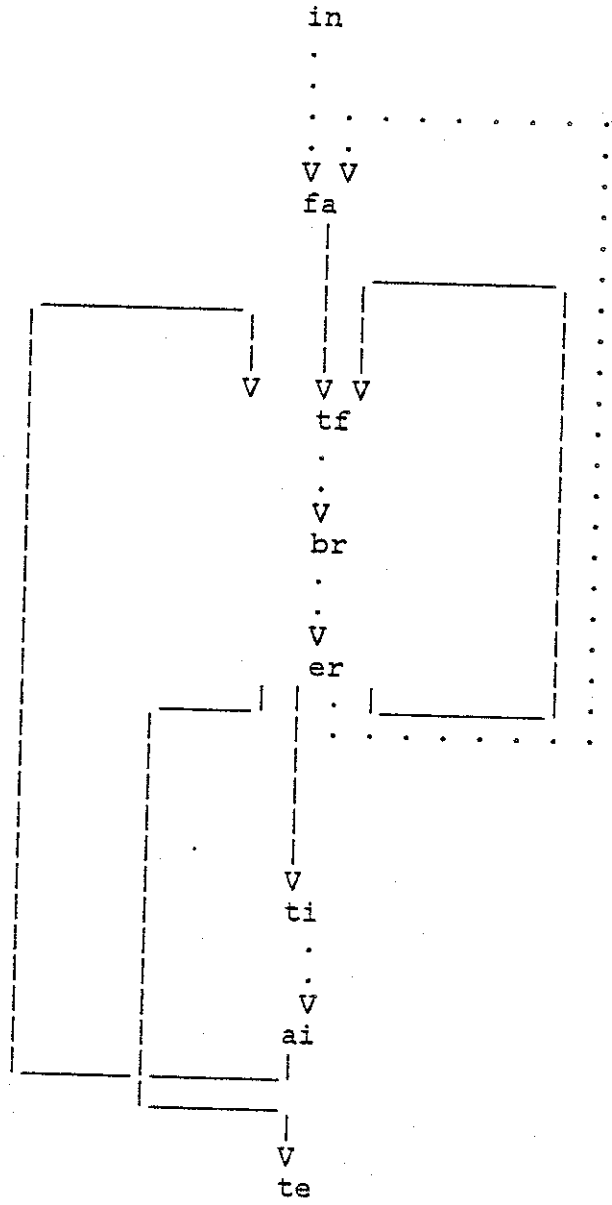
The graph of Figure 5 admits to significant simplification. The logic of the simplifications is summarized in Table 5 and the resulting simplified graph is presented as Figure 6.

Each simplification involves the deletion of an arc between a pair of AC nodes. The graph of Figure 5 was constructed using the algorithm of Section 4.3. Recall that an arc from $node_i$ to $node_j$ is intended to indicate that AC_i can cause AC_j to occur either in the same instant (if the arc is solid) or that AC_i schedules the occurrence of AC_j through a time-based signal (if the arc is broken). The Section 4.3 algorithm produces a graph which is not very "precise" since it includes many extra arcs.

To simplify this graph, in Table 5 note is taken of the values which are assigned to attributes in one AC and the values required to make the condition of another AC true. For example, Figure 5 has an arc form the initialization node, labeled "in," to the termination node, labeled "te." A simple analysis shows that the termination condition must be false after the actions of the initialization AC. Hence this arc is deleted in Figure 6. Similar logic is used to delete several other arcs.

6. Summary

Large, complex simulation modeling tasks, involving the collective talents of several persons, mandate the creation of model development environments. Prominent among the suite of software tools included in such an environment is the Model Generator, which assist the modeler's expression of concepts about the system under study, uninhibited by syntactic and semantic constraints of program execution. The condition specification (CS) is a target generative form that reveals the dynamic relationships inherent in discrete event models.



"___" Immediate change.
"..." Time delayed change.

Figure 6: Simplified Machine Repairman Action Cluster Incidence Graph

Also among the MDE tools is the Model Analyzer, which offers diagnostic assistance prior to the realization of an executable model representation -- a program. This paper focuses on graph-based diagnostic techniques applied to the CS. Beginning with the definition of three categories of diagnostic assistance, we have demonstrated how the CS admits the application of several techniques. The techniques are summarized within categories in Table 6.

The summary provided in Table 6 assists in emphasizing three points with regard to this research:

(1) The categorization of diagnostic assistance clarifies the extent to which the responsibility for detecting problems in a model specification is shared between the Model Analyzer and the modeler.

(2) While analytical diagnosis produces the most definitive statements about the model specification, informative diagnosis may present the highest utility/cost tradeoff, especially in the short term.

(3) Within the categories of comparative or informative diagnosis, precise measures and completely defined techniques cannot be recommended. Only by empirical investigation can proposed measures and techniques be evaluated.

Finally, the run-time graph can be derived from the "symbolic execution" of a condition specification. The concept of symbolic execution of a simulation model representation, that is not logically/arithmetically executable, represents an intriguing area of further research.

Category of Diagnostic Assistance	Properties, Measures, or Techniques Applied to the Condition Specification (CS)	Basis for Diagnosis
1) Analytical: Determination of the existence of a property.	a) Attribute Utilization: No attribute is defined that does not affect the value of another unless it serves a statistical (reporting) function.	Action Cluster Attribute Group (ACAG)
	b) Attribute Initialization: All requirements for initial value assignment to attributes are met.	ACAG
	c) Action Cluster Completeness: Required state changes within an action cluster are possible.	ACAG
	d) Attribute Consistency: Attribute Typing during model definition is consistent with attribute usage in model specification.	ACAG
	e) Connectedness: No action cluster is isolated.	Action Cluster Incidence Graph (ACIG)
	f) Accessibility: Only the initialization action cluster is unaffected by other action clusters.	ACIG
	g) Out-complete: Only the terminiation action cluster exerts no influence on other action clusters.	ACIG
	h) Revision Consistency: Refinements of a model specification are consistent with the previous version.	ACAG
2) Comparative: Measures of differences among multiple model representations.	i) Attribute Cohesion: The degree to which attribute values are mutually influenced.	Attribute Interaction Matrix (orginates with the ACAG)
	j) Action Cluster Cohesion: The degree to which action clusters are mutually influenced.	Action Cluster Interaction Matrix (originates with the ACAG)
	k) Complexity: A relative measure for the comparison of a CS to reveal differences in specification (clarity, maintainability, etc.) or implementation (run-time overheads) criteria.	ACIG
3) Informative: Charateristics extracted or derived from model representations.	l) Attribute Classification: Identification of the function of each attribute (e.g. input, output, control, etc.)	ACAG
	m) Precedence Structure: Recognition of sequential relationships among action clusters.	ACIG
	n) Decomposition: Depiction of subordinate relationships among components of a CS.	ACIG
	o) Run-time Graph: A representation of the "symbolic execution" of the CS.	Run-time Graph

Table 6. Categorized Summary of Diagnostic Assistance

BIBLIOGRAPHY

BALCO83 Balci, Osman, "Requirements for Model Development Environments," Technical Report CS83022-R, Department of Computer Science, Virginia Tech, Blacksburg, September 16, 1983.

CLEMA78 Clementson, A. T., *Extended Control and Simulation Language/ Computer Aided Programming System, Detailed Reference Manual*, The University of Birmingham, Birmingham, England, April 1978.

COXD61 Cox, D. R., and W. L. Smith, *Queues*, Methuen and Company, Ltd., 1961.

CURTB79 Curtis, B., S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," *IEEE Transactions of Software Engineering*, Vol. SE-5, No. 2, March, 1979, pp. 96-104.

DAVIN76 Davies, N.R., "A Modular Interactive System for Discrete Event Simulation Modeling," Proceedings Ninth Hawaii International Conference in System Sciences, Western Periodical Company, January 1976.

DECAR76 DeCarvalho, R. S., and J. G. Crookes, "Cellular Simulation," *Operational Research Quarterly*, Vol. 29, No. 1, 1976, pp. 31-40.

ELSHJ78 Elshoff, J. L., and M. Marcotty, "On the Use of the cyclomatic Number to Measure Program Complexity," *SIGPLAN Notices*, Vol. 13, No. 12, December 1978, pp. 29-40.

HENRS81a Henry, Sallie, and Dennis Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transaction on Software Engineering*, Vol. SE-9, No. 5, September 1981, pp. 510-518.

HENRS81b Henry, Sallie, and Dennis Kafura, "On the Relationships Among Three Software Metrics," Proceedings 1980 Symposium Software Quality Assurance, in *Performance Evaluation Review*, Vol. 10, No. 1, 1981, pp. 81-88.

HOWDW82 Howden, William E., "Contemporary Software Development Environments," *Communications of the ACM*, Vol. 25, No. 5, May 1982, pp. 318-329.

MATHS75 Matthewson, S. C., "Interactive Simulation Program Generators," *Simulation '75*, M. H. Hamze, ed., Acta Press, Calgary, Alberta, 1975.

MCCAT76 McCabe, T. J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308-320.

NANCR71 Nance, Richard E., "On Time Flow Mechanisms for Discrete Event Simulation," *Management Science*, Vol. 18, No. 1, September 1971, pp. 59-93.

NANCR81a Nance, Richard E. "Model Representation in Discrete Even Simulation: The Conical Methodology," Technical Report CS81003-R, Department of Computer Science, Virginia Tech, Blacksburg, March 15, 1981.

NANCR81b Nance, Richard E., Ahmed L. Mazaache, and C. Michael Overstreet, "Simulation Model Management: Resolving the Technological Gaps," Proceedings Winter Simulation Conference, Atlanta, GA, December 1981, pp. 173-179.

NANCR83 Nance, Richard E., "A Tutorial View of Simulation Model Development," Proceedings Winter Simulation Conference, Arlington, VA, December 1983, pp. 325-331.

NEWEA72 Newell, Allen, and Herbert A. Simon, *Human Problem Solving*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

ORENT82 Oren, Tuncer I., "Computer Aided Modelling Systems," in *Progress in Modelling and Simulation*, F. E. Cellier, ed., Academic Press, London, 1982.

OVERC82 Overstreet, C. Michael, "Model Specification and Analysis for Discrete Event Simulation," Ph.D. Dissertation, Department of Computer Science, Virginia Tech, Blacksburg, December 1982.

OVERC83 Overstreet, C. Michael, and Richard E. Nance, "A Specification Language to Assist in Analysis of Discrete Event Simulation Models," Technical Report CS83026-R, Department of Computer Science, Virginia Tech, Blacksburg, September 30, 1983.

PALMD47 Palm, D. C., "The Distribution of Repairmen in Servicing Automatic Machines," (Swedish), Industritidningen Norden, Vol. 175, p. 75.

SALTG68 Salton, Gerald, *Automatic Information Organization and Retrieval*, McGraw-Hill, New York, NY, 1968.

SCHRL83 Schruben, Lee, "Simulation Modeling With Event Graphs," *Communications of the ACM*, Vol. 26, No. 11, November 1983, pp. 957-963.

VIDAC80 Vidallon, C., "GASSNOL: A Computer Subsystem for the Generation of Network Oriented Languages with Syntax and Semantic Analysis," *Simulation '80*, Interlaken, Switzerland, June 25-29, 1980.

WASSA81 Wasserman, A. I., *Tutorial: Software Development Environments*, IEEE Computer Society Press, MD, 1981.

WOODM79 Woodward, M. R., M. A. Hennell, and D. Hedley, "A Measure of Control Flow Complexity in Program Text," *IEEE Transactions of Software Engineering*, Vol. SE-5, No. 1, January 1979, pp. 45-50.

ZEIGB84 Zeigler, Bernard P., *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, New York, 1984.