THE MODEL ANALYZER: PROTOTYPING THE DIAGNOSIS OF DISCRETE-EVENT SIMULATION MODEL SPECIFICATIONS

by

Frederick Anthony Puthoff

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Richard E. Nance, Chairman

James D. Arthur

Osman Balci

September, 1991

Blacksburg, Virginia

THE MODEL ANALYZER: PROTOTYPING THE DIAGNOSIS OF DISCRETE-EVENT SIMULATION MODEL SPECIFICATIONS

by

Frederick Anthony Puthoff

Committee Chairman: Richard E. Nance
Computer Science

(ABSTRACT)

This thesis describes the development of a Model Analyzer prototype for the Simulation Model Development Environment that analyzes the specifications of a discrete-event simulation model. The Model Analyzer provides early feedback by operating on model specifications instead of waiting until an executable version is produced. Analysis of model specifications allows the modeler to detect errors early in the life-cycle and provides information about the model that may prove helpful in the verification, validation, and eventual translation of the model.

With the help of a graphical user interface, the Model Analyzer provides automated and semi-automated support to the modeler. The Condition Specification, a world-view independent specification, of a model is parsed and stored in a relational database. Using graph-based diagnostic techniques, the Model Analyzer presents graphical representations of the model and performs analytical, comparative, and informative diagnosis. Also, an expert system is developed that uses generic simulation knowledge to simplify one graphical representation of the model.

Based on the accomplishment of the design objectives, an evaluation of the Model Analyzer is conducted. Results from the expert system and Model Analyzer are provided that indicate the Model Analyzer and the expert system are helpful analysis tools.

Acknowledgements

I wish to acknowledge my adviser, Dr. Richard E. Nance, for all the intellectual conversations that provided valuable insight and guidance to my research. I also acknowledge Dr. Osman Balci for serving on my committee and giving me a desk and office space to conduct this research, and Dr. James D. Arthur for his willingness to be a part of my committee. I thank Trish Hubble and Sandra Griffith for their kindness in the thousand times I asked to borrow the keys to the copy room and asked for office supplies. I also acknowledge Ernie Page for providing insight to his previous work and for suffering through a tough Cincinnati Reds baseball season with me. I especially thank Joe Derrick for his friendship and the countless times he helped me with the computer system. Most importantly, I wish to thank my wife, Hyesuk, for her total support and confidence in me, and for her help cutting and pasting the many figures throughout this thesis. Last of all, I wish to thank the United States Army for allowing me the opportunity to pursue my Master of Science degree.

Table of Contents

List of Illustrations

List of Tables

List of Acronyms

AACM	—	Attribute Action Cluster Matrix
AC	—	Action Cluster
ACAG	—	Action Cluster Attribute Graph
ACAM	—	Action Cluster Attribute Matrix
ACIG	—	Action Cluster Incidence Graph
ACIM	—	Action Cluster Interaction Matrix
AIM	—	Attribute Interaction Matrix
BNF	—	Backus-Naur Form
CAP	—	Condition Action Pair
CAS	—	Credibility Assessment Stage
CAT	—	Control and Transformation Metric
CM	—	Conical Methodology
CS	—	Condition Specification
DBMS	—	Database Management System
EQUEL/C	—	Embedded Query Language for INGRES in C
INGRES	—	Interactive Graphics and Retrieval System
Lex	—	Lexical Analyzer
MA	—	Model Analyzer
MMDE	—	Minimal Model Development Environment
QUEL	—	Query Language for INGRES
SMDE	—	Simulation Model Development Environment
SunView	—	Sun Visual/Integrated Environment for Workstations
UAN	—	User Action Notation
UIMS	—	User Interface Management System
Yacc	—	Yet Another Compiler-Compiler

1. Introduction

The purpose of studying real world systems through modeling is to aid in the analysis, understanding, operation, prediction, or control of the systems without constructing or operating the real thing [Neelamkavil 1987, p. 1]. The use of a simulation model is helpful when a system does not exist or experimentation with the system is expensive or inappropriate [Davies and O'Keefe 1989, p. 1].

The Department of Defense recently has increased its use of models and simulation to support the development, test, and evaluation process for military systems [Horowitz 1990]. With the declining defense budget, experimental testing of military systems is likely to be reduced because of the prohibitive costs. Consequently, simulation testing becomes more important. The Department of Defense [1989] has included software producibility as well as simulation and modeling as two critical technologies. Critical technologies are defined as technologies with great promise of ensuring the long-term superiority of the United States weapon systems.

Although the use of models is less expensive than experimentation with the actual systems, model development and simulation also can be very costly. One way to reduce the cost is to create an environment to assist the model development process. Interest in support environments has permitted a shift in representation focus from program to model. Modeling and model representation has become a more visible simulation activity.

An objective of the research in the **Simulation Model Development Environment (SMDE) Project** [Balci and Nance 1987a] at Virginia Polytechnic

Institute and State University is to assist the model development process. The SMDE is an integrated and comprehensive collection of computer-based tools that assist the modeler throughout the development of a discrete-event simulation model. The SMDE is composed of four layers. The *Minimal Model Development Environment (MMDE)* layer provides a set of "minimal" tools for the development and execution of a model. One tool within this layer is the **Model Analyzer**, which is the focus of this research more fully explained in Section 2.2.

1.1 Importance of the Model Analyzer

A modeler needs feedback on the "goodness" of the modeling effort: (1) Is the model representation correct? (2) How complex is the representation? (3) Does the model appear to be easily extensible? The objective of the Model Analyzer is to provide this early feedback by operating on model specifications created by the model generator instead of waiting until an executable version is produced by the model translator.

This analysis of model specifications is important for two reasons. First, model diagnosis allows one to correct errors in the model or its definition early in the life-cycle, therefore reducing the cost and time of development. Secondly, diagnosis provides information about the model that may be helpful in the verification, validation, and eventual translation of the model.

1.2 Organization of Research

The research in this thesis describes the development of a Model Analyzer prototype. This prototype utilizes knowledge gained from an earlier prototype developed in 1985 [Moose and Nance 1987,1989]. The early prototype is explained in Section 3.2. The current prototype represents a significant extension of functionality.

Chapter 2 reviews the literature concerning life-cycles used to develop simulation models, Simulation Model Development Environment Project, human-computer interfaces, Condition Specification, and model analysis. The role of model analysis is discussed in Chapter 3. In Chapter 4, the design of the Model Analyzer prototype is fully described. An expert system approach to simplifying the Action Cluster Incidence Graph (ACIG) is revealed in Chapter 5. An example is presented in Chapter 6. Chapter 7 offers an evaluation of the Model Analyzer prototype. Finally, Chapter 8 presents the conclusion and possible future research options.

2. Literature Review

Before the Model Analyzer prototype is described, a review of related topics is needed to understand the basis for the prototype. Section 2.1 discusses three life-cycles used to develop simulation models. The Simulation Model Development Environment (SMDE) Project is discussed in Section 2.2. Human-computer interface principles are described in Section 2.3, and a type of specification language is explained in Section 2.4. The final section describes some previous work done on model analysis.

2.1 Life-Cycles Used to Develop Simulation Models

All developing systems start with some statement of need (requirements) and hopefully end with a protracted period of use (operation or experimentation). The description of the management process between these two points is termed a life-cycle model [Blum 1982, p. 18]. Each life-cycle must contain some form of three phases: definition, development, and maintenance [Pressman 1987].

Life-cycles serve two primary functions. First, they determine the order of the stages involved in software development and evolution. Secondly, life-cycles establish the transition criteria for progressing from one stage to the next [Boehm 1986, p. 14].

Today, many different life-cycles exist for the development of simulation models. Here, two software life-cycle models and a discrete-event simulation life-cycle are presented: the generic software life-cycle model, the operational approach, and the discrete-event simulation life-cycle model. These three are among the most commonly used approaches to develop simulation models.

2.1.1 Generic Software Life-Cycle Model

The generic software life-cycle model can be used to develop simulation models as well as other types of software. Sometimes, this life-cycle model is called the *waterfall model,* because the model is composed of a sequence of tasks that "waterfall" into each other [Overstreet et al. 1986].

The tasks or phases of the generic software life-cycle model are shown in Table 1, which is taken from [Barger 1986]. Most other life-cycles are a variation of this generic life-cycle.

2.1.2 Operational Approach

Zave [1984a] developed the operational approach of developing software that also may be used to develop simulation models. This approach, shown in Figure 1, is based on the separation of problem-oriented from implementation-oriented concerns. This separation is achieved by developing a problem-oriented, executable specification and a transformational implementation of the proposed problem.

During the *specification phase*, a complete and executable representation of the proposed system, called the *operational specification*, is formed. In the *transformational phase*, the specification is subject to transformations that preserve the behavior of the system, but alter the way the behavior is produced. The goal is to automate these transformations, which yield a fully specified, executable structure called the *implementation-oriented specification.* The last phase maps the transformed specification into an implementation language.

What are the advantages of using the operational approach over the generic life-cycle model? In the operational approach, the specifications are formal and rigorous, thus allowing them to be formally analyzed. The specifications can be produced

Table 1. Phases of a Generic Software Life-Cycle Model [Barger 1986, p. 16]

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

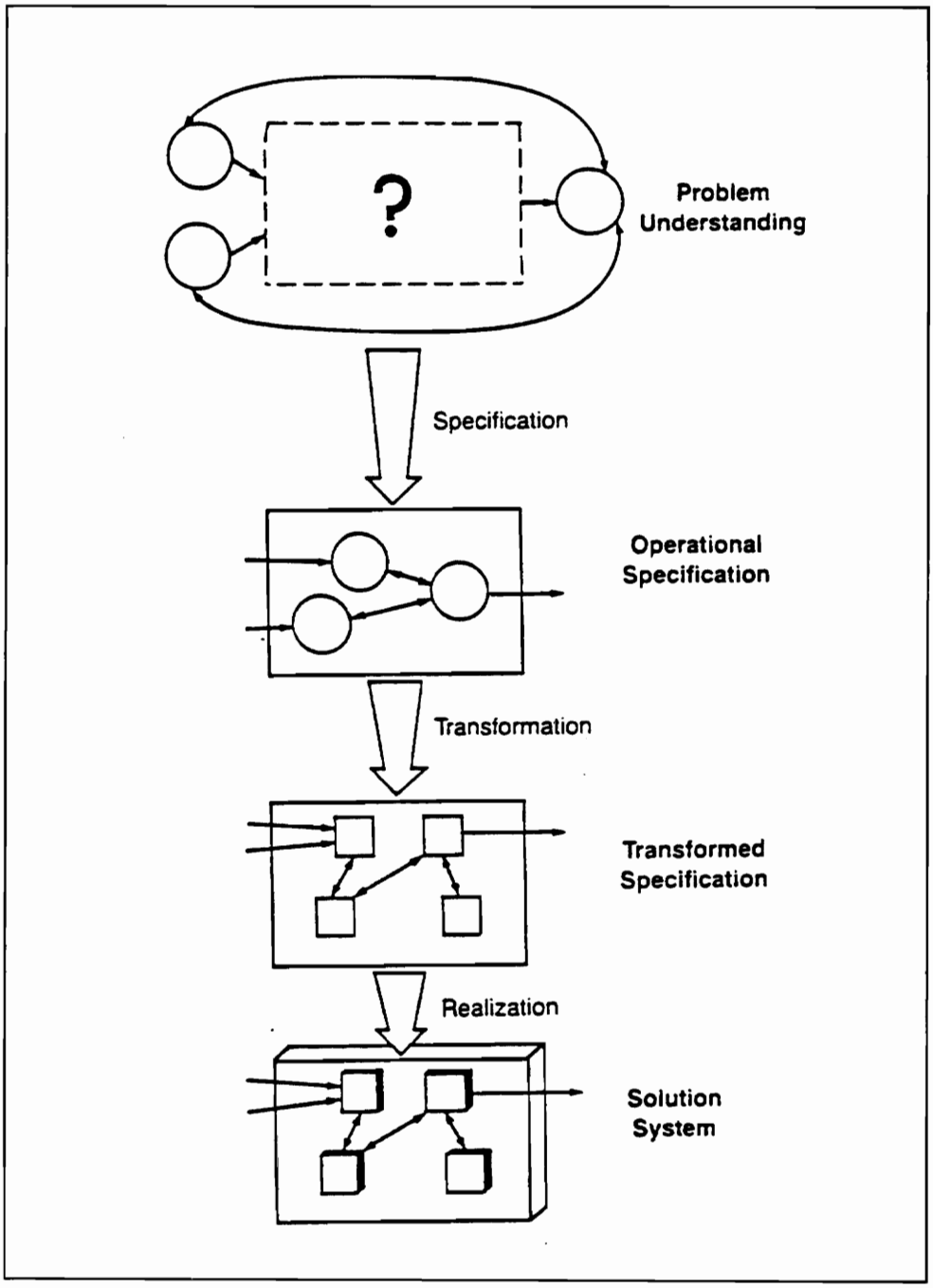


Figure 1. The Operational Approach [Zave 1984a, p. 108]

automatically with the help of rapid prototyping. The software process is easily verified by following the transformations of the specification into the implementation. Maintenance is performed by modifying the specifications, which then are reapplied to produce a modified implementation. A management advantage is that an executable specification or executable version of the system is produced early in the software development process.

The operational approach also has its weaknesses. Premature design decisions may be made. Also, the operational specifications usually perform poorly by running slowly. The transformational implementation is new and only beginning to be developed. The formal specifications are not easily understood, thus the users may find it difficult to choose and apply the right transformations.

2.1.3 Discrete-Event Simulation Life-Cycle Model

Nance and Balci have developed a detailed simulation model life-cycle, shown in Figure 2, that outlines the development of discrete-event simulation models [Balci 1987a, b, 1990]. A discrete-event simulation involves the modeling on a digital computer of a system in which state changes can be represented by a collection of distinct events [Fishman 1973, p.23].

The discrete-event simulation life-cycle model is composed of ten phases, ten processes, and thirteen credibility assessment stages (CASs). Each oval symbol in the diagram represents a phase. The dashed arrows represent processes that relate the phases to each other, and the solid arrows depict the CASs. The life-cycle is not strictly sequential as might be implied by the dashed arrows. Instead, the cycle is iterative in nature in which reverse transitions are expected. An error may be identified that may cause one to return to an earlier process and start over [Overstreet et al. 1986].

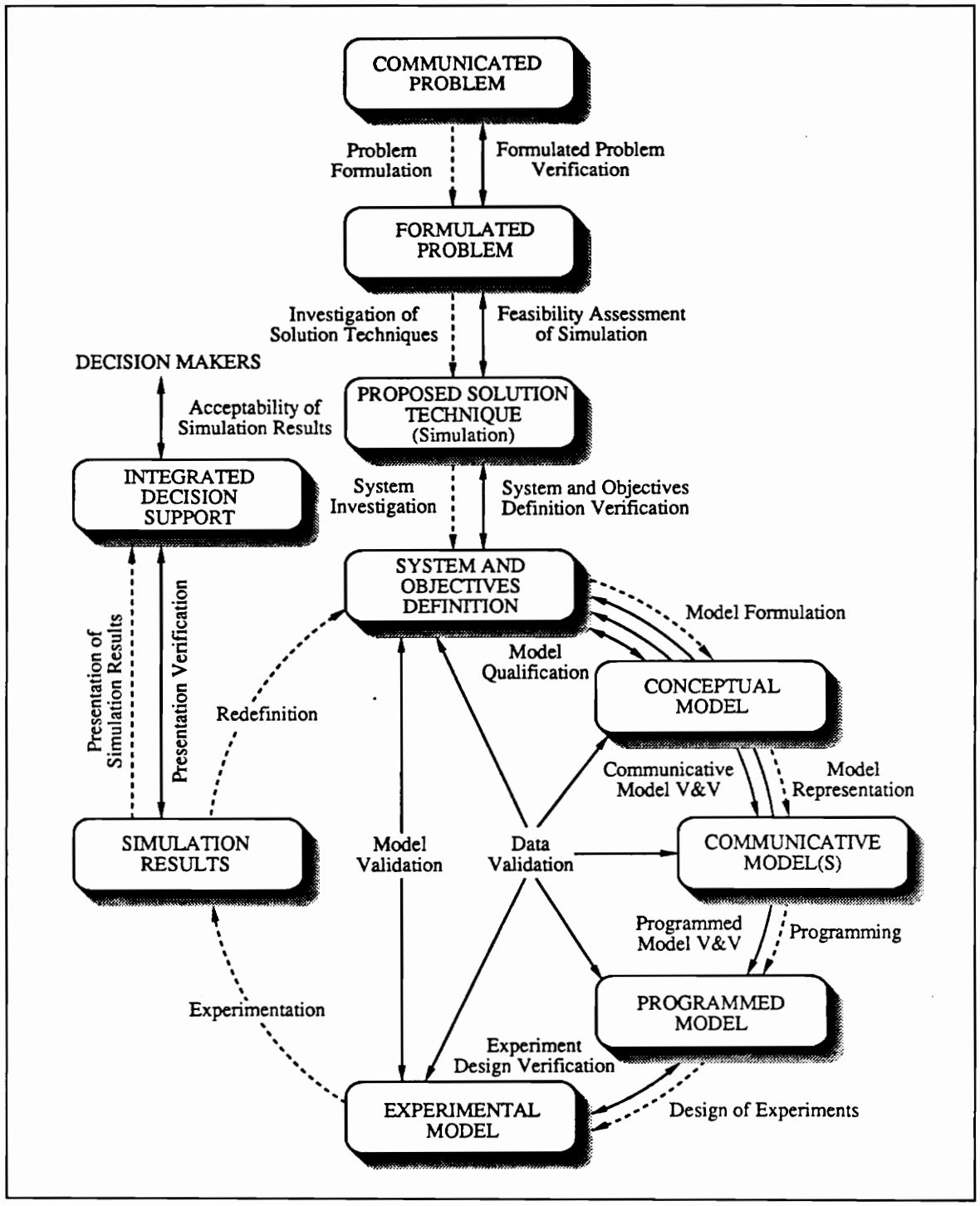


Figure 2. Discrete-Event Simulation Life-Cycle Model [Balci 1990, p. 26]

The simulation life-cycle model starts with the Communicated Problem phase. The communicated problem is developed when a client recognizes a problem and communicates that problem to an analyst. The last process in the simulation life-cycle is Presentation of Results, where the results are interpreted and presented to decision makers for their approval or disapproval.

The CAS important to this research is Communicative Model Verification and Validation, which confirms the adequacy of the communicative model to provide an acceptable level of agreement for the domain of the intended application. This verification may be accomplished by using one or more of the following techniques: desk checking, model review, graph-based analysis, instrumentation-based testing, and functional testing [Balci 1990].

2.2 Simulation Model Development Environment Project

The **Simulation Model Development Environment (SMDE) Project** at Virginia Polytechnic Institute and State University can be characterized as a simulation support environment or a computer-aided simulation engineering environment. The SMDE is an integrated and comprehensive collection of computer-based tools that assist the modeler throughout the development process of creating a discrete-event simulation model [Balci and Nance 1987a]. Since 1983, the project has progressed to its current development shown in Figure 3.

The objectives of the SMDE [Balci 1986] are to:

(1) offer cost-effective integrated and automated support of model development throughout its entire life-cycle,

(2) improve the model quality by effectively assisting in the quality assurance of the model,

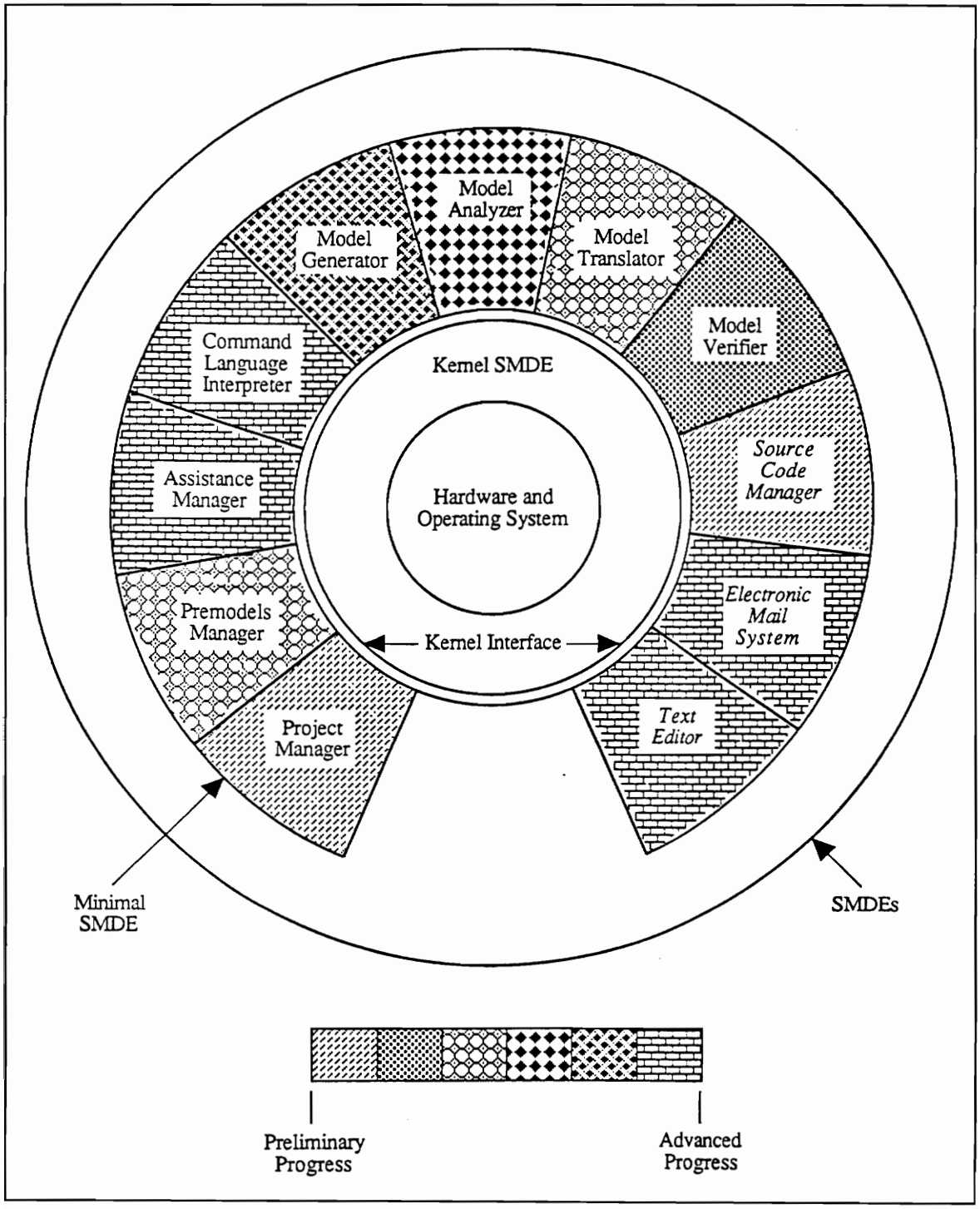


Figure 3. The Simulation Model Development Environment [Balci et al. 1990, p. 258]

(3) increase significantly the efficiency and productivity of the product team, and

(4) decrease substantially the model development time.

To achieve these objectives, the SMDE is composed of four layers: (1) Hardware and Operating System, (2) Kernel SMDE, (3) Minimal SMDE, and (4) SMDEs. The Minimal SMDE layer provides a "comprehensive" set of tools that are "minimal" for the development and execution of a model. One of these tools, the Model Analyzer, is the subject of this thesis.

2.3 Human-Computer Interfaces

A *Human-computer interface*, or more commonly called a *user interface*, is the software and hardware through which an exchange of symbols and actions between a human and computer occurs [Hartson and Hix 1989, p. 8]. Having a good human-computer interface is essential in developing good software applications or prototypes. A poor user interface will make a software system unusable and unsuccessful no matter how good of quality the rest of the software is. A problem with designing human-computer interfaces is that developers know better how to build a system than how to specify what it is to accomplish or how it is to interact with the user. Even applying software engineering techniques to the production of human-computer interfaces does not always produce usable interfaces [Hartson et al. 1990, p. 181].

To develop a good human-computer interface, proper design principles, which are summarized from [Page 1990], must be followed. Dumais [1982] studied whether an "easy to use" software system should be menu-selection, keyword, or natural-language based. Their results indicated natural-language based systems are easiest to use but also the most difficult to develop. With all design decisions, tradeoffs must be made. A system should be simple but powerful and easy to learn but appealing to the experienced

users. An algorithm for the perfect human-computer interface does not exist, but certain common principles are found in the literature. According to Hanson [1971], Wasserman [1973], and [Gaines 1975], all state the user is important. Also, each designer stresses consistency and flexibility, as well as explicit on-line assistance and diagnostics. A good interface also must be robust and flexible to provide for a wide range of users. Procedures should be simple and easy to learn, but also allow "shortcuts" for experienced users.

A major distinction between software and interface design is that software design is system-centered; whereas, interface design is user-centered. While designing a human-computer interface, one must focus on the user and what the user perceives while performing tasks with the computer. If the developer can integrate behavior techniques into the constructional development process, then this creates a more effective method of creating useful and usable human-computer interfaces. Behavior tools are needed to enhance the interface development process. One such tool is *User Action Notation (UAN)*, a task- and user-oriented notation for behavior representations of asynchronous, direct manipulation interface designs [Hartson et al. 1990]. UAN, which is described in greater detail in Section 4.3.1, is used to develop the human-computer interface of the Model Analyzer prototype.

Another paper that emphasized the importance of human-computer interface development is [Hartson and Hix 1989]. This article addresses the problem of not how to construct good interfaces, but how to provide an environment in which good interfaces can be constructed. Examples and important concepts of user interface management systems (UIMSs) are also presented.

2.4 Condition Specification

After a discrete-event simulation model is defined, the model must be precisely defined in a model specification language. This approach helps reduce the modeling costs by interposing an intermediate form between the Conceptual Model and the Communicative Model, or executable representation of the model [Overstreet and Nance 1985].

A model specification is defined as a quintuple:

< input specifications,
output specifications,
object definition set,
indexing attribute,
transition specification >.

The *input* and *output specifications* form the interface between the model and its environment. The *object definition set* contains the attributes associated with each object. The attribute "system time", is usually the *indexing attribute* that allows for temporal relationships within the model. Finally, the *transition specification* defines the initial state, the termination conditions, and the definition of the dynamic structure of the model [Overstreet and Nance 1985].

To perform model diagnosis on its specifications, one needs to have a "formal" specification of the model. One such "formal" model specification language is the **Condition Specification (CS)** [Overstreet and Nance 1985]. The CS is a primitive specification language that contains no semantics or syntactic elements needed for executable code. The CS is also world-view independent and explicitly describes objects and the relations among objects to define the model. An advantage of the condition specification is that it can be analyzed to detect errors and provide over useful information to the modeler.

Three components compose each model condition specification: (1) system interface specification, (2) specification of model dynamics, and (3) report specification. These three parts are explained in the following sections. Table 2 shows the syntax of the CS primitives [Overstreet and Nance 1985].

2.4.1 System Interface Specification

The system interface specification, referred as "interface specification" by Overstreet and Nance [1985], identifies the input and output attributes by name, data type, and communication type (input or output). Every CS must include at least one output attribute. An example of the system interface specification, described in Nance and Overstreet [1987b], for the Single Server Queue Model is shown in Figure 4. The Single Server Queue represents a system that contains a single server that services incoming parts, which wait in a single, first come, first fixed, infinite queue before they are serviced. Ater each part is serviced, it leaves the system.

2.4.2 Specification of Model Dynamics

The specification of model dynamics is broken into two major components: object specification and transition specification. The following two sections explain these components.

2.4.2.1 Object Specification

The object specification names each object in the model and its associated attributes. Each attribute is typed in a fashion similar to Pascal. The types include integer, real, Boolean, character, string, enumerated (shown by a list of values), and a special type called signal. The signal attributes are time-based and dependent upon the

Table 2. Syntax and Function of Condition Specification Primitives

NAME	SYNTAX	FUNCTION
Value change description	<attribute_name> := <attribute_expression>	Assign attribute values
Set Alarm	SET ALARM (<alarm_name>, <alarm_time> [, <parameter_list>])	Schedule an alarm
Cancel Alarm	CANCEL ALARM (<alarm_name> [, <parameter_list>])	Cancel scheduled alarm
When Alarm	WHEN ALARM (<alarm_name_exp> [, <parameter_list>])	Time sequencing condition
After Alarm	AFTER ALARM (<alarm_name_exp> AND <Boolean_exp> [, <parameter_list>])	Time sequencing condition
State-based Condition	WHEN <Boolean_expression>	Boolean condition
Create	CREATE (<object_type> [, <object_type>]*)	Generate new object
Destroy	DESTROY (<object_type> [, <object_type>]*)	Eliminate an object
Input	INPUT (<attribute_name> [, <attribute_name>]*)	Assign value via input
Output	OUTPUT (<attribute_name> [, <attribute_name>]*)	Produce output
Start Condition	START	Begin simulation
Stop	STOP	End simulation
Comment	{ <any text not including a "}"> }	Comment

```
Input:
        parts.meanInterarrivalTime   {neg. exp. distribution mean}
                                     {  for interarrival times    } : nonnegative real;
        server.meanServiceTime       {neg. exp. distribution mean}
                                     {  for service times         } : nonnegative real;
        system.stopNum               {number serviced before      }
                                     {  termination               } : positive integer;

Output:
        system.systemTime            {ending simulation time      } : nonnegative real;
```

Figure 4. Single Server Queue System Interface Specification

the system time, which allows the modeler to describe the time sequencing of the model. Figure 5 shows an example of the Single Server Queue object specification.

2.4.2.2 Transition Specification

The transition specification defines the dynamic structure of a model and consists of a set of ordered pairs, called *condition action pairs (CAPs)*. Each CAP contains a condition and an action. A condition is either (1) a Boolean expression preceded by the CS primitive WHEN, (2) a time sequencing condition using the primitives WHEN ALARM or CANCEL ALARM, or (3) a begin simulation condition called START. Each action performs one of the following functions: (1) assigns a value to an attribute by the assignment operator ":=", (2) schedules an alarm with the SET ALARM primitive, (3) generates new objects by the primitive CREATE, (4) destroys objects with the DESTROY primitive, (5) assigns an input value to an attribute by using the INPUT command, (6) produces output with the OUTPUT command, (7) ends the simulation with the primitive STOP, or (8) calls a function. When a condition of a CAP becomes true, the associated action is performed. It is possible for more than one condition to be true at any instance in time. Two CAPs must be present in every model transition specification: the *initialization* CAP, which contains the START condition that is true only at the beginning of the simulation, and the *termination* CAP, which contains the STOP action that terminates the simulation.

Usually, condition action pairs with identical conditions are grouped together to form *action clusters (ACs)*. Each AC is then given a unique name. The action clusters consist of one condition and many actions. If functions are used in the transition specification, a *function specification* is given that defines the purpose of each function

```
OBJECT_SPEC single_server_queue

    OBJECT system
        systemTime:                 nonnegative real;
        stopNum:                    positive integer;

    OBJECT server
        serverStatus:               (busy, idle);
        meanServiceTime:            nonnegative real;
        endService:                 signal;
        numServed:                  nonnegative integer;

    OBJECT parts
        numWaiting:                 nonnegative integer;
        meanInterarrivalTime:       nonnegative real;
        arrival:                    signal;
```

Figure 5. Single Server Queue Object Specification

as well as their inputs and outputs. Figure 6 shows the transition specification for the Single Server Queue Model.

2.4.3 Report Specification

The report specification contains actions that are performed during a simulation run and an analysis routine for the data produced during the simulation run. The data is written to a file. Overstreet [1982 p. 90] does not define the syntax for a report specification and separates the report specification from the condition specification. Overstreet states this is not mandatory but often desirable.

2.5 Model Analysis

The ability to analyze a discrete-event simulation model before it is translated into an executable form is an important step in the understanding of that model. Model analysis has been accomplished by many different methods or approaches. Three approaches are outlined in this paper: (1) M-labeled digraphs by Burns and Winstead [1985], (2) event graphs by Schruben [1983], and (3) graph-based analysis based on the *action cluster attribute graph (ACAG)* and *action cluster incidence graph (ACIG)* originally developed by Overstreet [1982]. The first two techniques are described in the following two sections; whereas, the latter technique is explained in Chapter 3.

2.5.1 M-Labeled Digraphs

The method of analyzing models with M-labeled digraphs [Burns and Winstead 1985] is usually performed upon continuous simulation models, but also may be used upon discrete-event simulation models. Burns and Winstead's analysis is a type of

```
TRANSITION_SPEC single_server_queue

    AC: Initialization
        WHEN START:
            system.systemTime := 0;
            server.serverStatus := idle;
            parts.numWaiting := 0;
            server.numServed := 0;
            INPUT (parts.meanInterarrivalTime, server.meanServiceTime,
                system.stopNum);
            SET ALARM (parts.arrival, 0);

    AC: Termination
        WHEN server.numServed >= system.stopNum:
            OUTPUT (system.systemTime);
            STOP;

    AC: Arrival
        WHEN ALARM (parts.arrival):
            parts.numWaiting := parts.numWaiting + 1;
            SET ALARM (parts.arrival, negexp(parts.meanInterarrivalTime));

    AC: Begin_Service
        WHEN parts.numWaiting > 0  AND  server.serverStatus = idle:
            parts.numWaiting := parts.numWaiting - 1;
            server.serverStatus := busy;
            SET ALARM (server.endService, negexp(server.meanServiceTime));

    AC: End_Service
        WHEN ALARM (server.endService):
            server.numServed := server.numServed + 1;
            server.serverStatus := idle;
```

Figure 6. Single Server Queue Transition Specification

geometric approach, often called "structural analysis", concerning systemic behavior as it relates to a model structure.

Ordinary digraphs are permitted only one edge between each pair of nodes; whereas, M-labeled digraphs may have as many edges as there are labels between any vertex pair. Therefore, M-labeled digraphs can have *m* distinct "signs" to differentiate among *m* distinct edges. A separate adjacency matrix is used for each label in M. The technique is concerned with counting the number of sequences of a particular type or "sign" between a pair of vertices in the M-labeled digraph. This information is usually stored in matrix form that allows for computer analysis of the information.

Burns and Winstead [1985, p. 356] show how a digraph can be used to represent activity/event characterizations with an example of a neighborhood grocery store. The edges of the graph represent activities; the nodes represent events; and the labels signify the statistical distributions that characterize time-durations of activities.

The authors also state M-labeled digraphs appear to extend the potential use of signed digraphs as an analytical tool; however, some problems exist. Meaningful interpretation of results is difficult with large label sets, and careful development of the important Calley multiplication table for the groupoid requires much time and effort.

2.5.2 Event Graphs

An event graph, introduced by Schruben [1983], is a graphical technique for event-oriented systems that is based on the dual time-state relationship of system events. Event graphs are constructed as follows:

(1) System events are defined and numbered.

(2) Events are defined by the transformation of the system state variables that take place when each event occurs.

(3) Event vertices on a directed graph represent system state changes.

(4) S_i represents the set of state variables possibly altered by event i.

(5) Directed edges connect the events and indicate how events influence the occurrence of another event.
- solid line (event-scheduling edge)
- dashed line (event-cancelling edge)

(6) Edges also have attributes of delay time and conditions related to event instigation.

Events are usually programmed as separate routines or procedures in an event-scheduling simulation. Schruben gives two examples using event graphs: machines with service interruption model and a semiautomatic machine model.

Event graph analysis can aid (1) in identifying state variables, (2) in determining what events must be initially scheduled, (3) in anticipating possible logic errors due to simultaneous events, and (4) in eliminating unnecessary event routines prior to coding a simulation. Schruben uses five rules to perform the above four tasks.

A later paper by Som and Sargent [1989] also examines event graphs, focusing on the third and fourth tasks noted above. To show the hierarchy and possible grouping of events as a single unit without violating execution order priorities, they introduce three concepts: (1) expanded event graphs, (2) super events, and (3) execution order priority matrix. An expanded event graph is a visual way of showing the structural elements of a model within its hierarchy, and represents a very useful starting point for the implementation of a model. An expanded event graph is made up of super events, which each consist of one root event and its associated subgraph of events. An execution order priority matrix is a triangular matrix that defines the execution order priority for every pair of events that may interact and occur simultaneously.

3. Role of Model Analysis

Analysis of the specification of a discrete-event simulation model plays a number of roles in the development process. First, analysis of the specification, instead of the executable code, provides an earlier detection of errors in the model that allows one to fix errors more easily and save on development costs. Second, model analysis can provide helpful implementation information not conveyed in the specifications alone. Certain run-time artifacts are inserted in the executable code (for execution purposes) that inhibit the consequent analysis of model specifications. Third, model analysis supports the automation-based paradigm, which advances the proposition of automatic transformation of the specifications into executable code. Since all the transformations are applied to the specifications, the specifications must be analyzed before the final code is developed. Fourth, model analysis is especially helpful to large projects by providing information that can be presented in an understandable fashion to all project participants. This helps improve the communication between teams and eases the development process.

Section 3.1 outlines the graph-based analysis approach originally developed by Overstreet [1982]. This approach forms the foundation to the analyzer prototype, which is the subject of this thesis. A previous analyzer prototype produced by Moose [Moose and Nance 1987, 1989] is described in Section 3.2. The last section presents the objectives of the current prototype.

3.1 Graph-Based Model Analysis

Overstreet [1982] first defined the graph-based model analysis, or also called model diagnosis, which is explicitly explained in [Nance and Overstreet 1986, 1987a, b]. Nance and Overstreet state "that model diagnosis should not require a significant investment of time and effort beyond that demanded of the modeler for specification and documentation of the model." Therefore, the graph-based representations used for analysis should be automatically derivable from the formal condition specification (CS), described in Section 2.4.

The two important digraphs used to analyze the model are the *Action Cluster Attribute Graph (ACAG)* and the *Action Cluster Incidence Graph (ACIG)*. Before explaining how to construct these two graphs, the following definitions, taken from Nance and Overstreet [1987a], are needed.

Control Attribute An attribute x is a *control attribute* of an action cluster if x appears in a condition expression of the action cluster.

Output Attribute An attribute x is an *output attribute* of an action cluster if the actions can change the value of attribute x.

Input Attribute An attribute x is an *input attribute* of an action cluster if the value of x affects the output attributes of the action cluster.

Table 3 shows the control, input, and output attributes for the Single Server Queue Model. This table serves to exemplify the ACIG and ACAG, which are described in the following two sections.

3.1.1 Action Cluster Attribute Graph

Nance and Overstreet [1987a, b] define the *Action Cluster Attribute Graph (ACAG)* for a CS as follows. Assume a condition specification with k time-based signal

Table 3. Single Server Queue Attribute Classification

Control Attributes	Input Attributes	Output Attributes
Action Cluster: Initialization		
		parts.arrival parts.meanInterarrivalTime parts.numWaiting server.meanServiceTime server.numServed server.serverStatus system.stopNum system.systemTime
Action Cluster: Termination		
server.numServed system.stopNum	system.systemTime	
Action Cluster: Arrival		
parts.arrival	parts.meanInterarrivalTime parts.numWaiting	parts.arrival parts.numWaiting
Action Cluster: Begin_Service		
parts.numWaiting server.serverStatus	parts.numWaiting server.meanServiceTime	parts.numWaiting server.endService server.serverStatus
Action Cluster: End_Service		
server.endService	server.numServed	server.numServed server.serverStatus

attributes, m other attributes, and n action clusters. Then G, a directed graph with $k + m + n$ nodes, is constructed as follows:

G has a directed, labeled 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.

Interactions in the ACAG that occur instantly are shown with a solid edge; whereas, time-delayed interactions are shown with a dashed line.

The ACAG displays the interactions between action clusters and attributes in the CS. The potential for actions of an AC to change the value of an attribute and the reverse (the influence of an attribute on the actions of an AC) are shown.

The ACAG for the Single Server Queue Model is shown in Figure 7. This ACAG has fourteen nodes and 26 edges. Even a very simple model such as the Single Server Queue has a cluttered ACAG. For this reason, the ACAG is usually represented as a matrix instead of a digraph. The ACAG matrix for the Single Server Queue is shown in Table 4.

One also can represent the ACAG as two separate Boolean matrices: the *attribute action cluster matrix (AACM)* and the *action cluster attribute matrix (ACAM)*. These two matrices are defined in [Nance and Overstreet 1987a, b] as follows. Let a CS have m action clusters ($ac_1, ac_2, \ldots, ac_m$), and n attributes ($a_1, a_2, \ldots, a_n$). The attribute action cluster matrix (AACM) for a CS is an n by m Boolean matrix in which:

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

The action cluster attribute matrix (ACAM) for a CS is an m by n Boolean matrix in which:

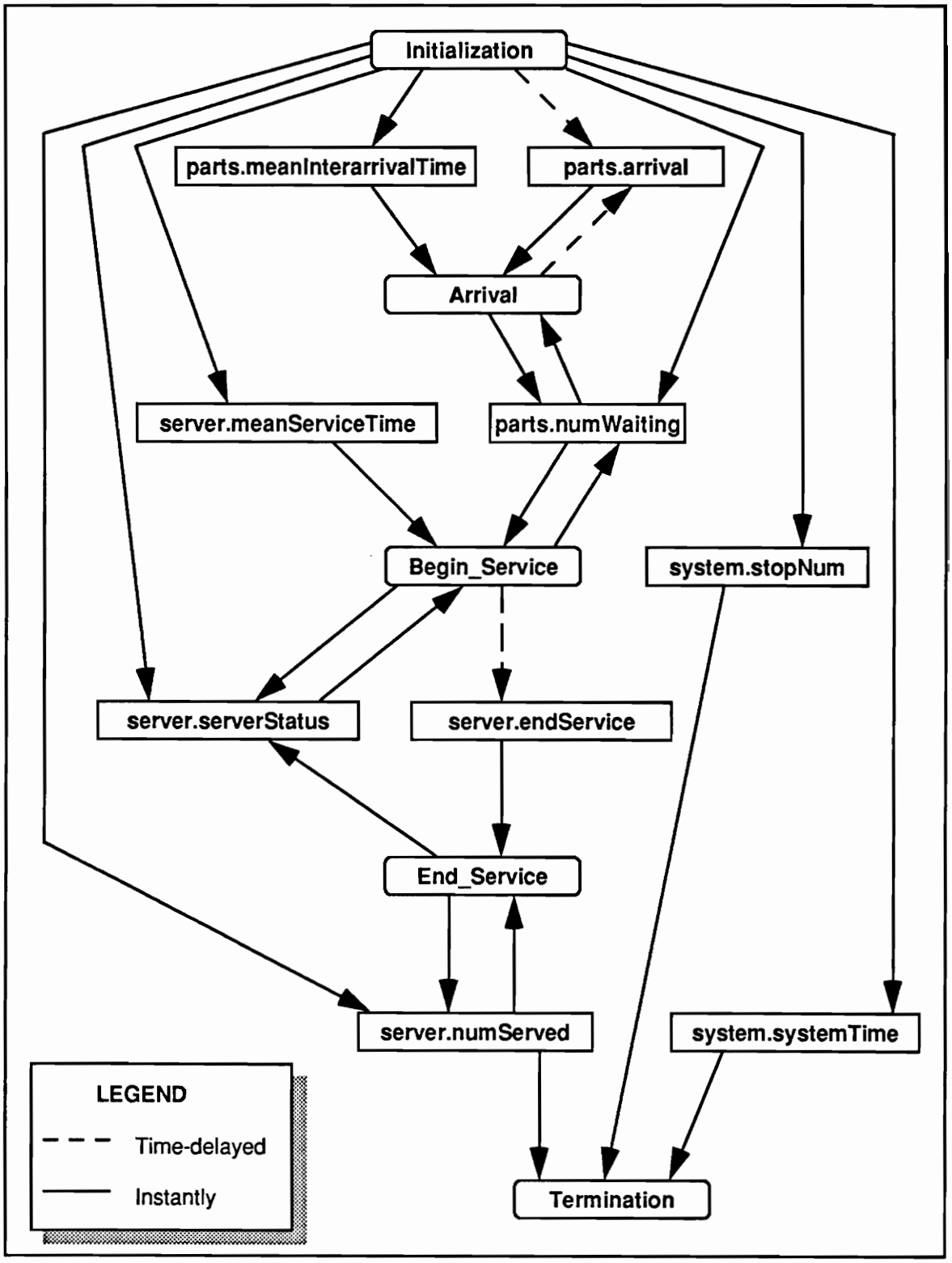


Figure 7. Single Server Queue Action Cluster Attribute Graph

Table 4. Single Server Queue Action Cluster Attribute Graph in Matrix Form

	Action Clusters				
Attributes	Arrival	Begin_Service	End_Service	Initialization	Termination
parts.arrival	C T			T	
parts.meanInterarrivalTime	I			O	
parts.numWaiting	I O	I C O		O	
server.endService		T	C		
server.meanServiceTime		I		O	
server.numServed			I O	O	C
server.serverStatus		C O	O	O	
system.stopNum				O	C
system.systemTime				O	I

LEGEND

I = Input O = Output

C = Control T = Time-delayed Output

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

From the AACM and ACAM, two other important matrices can be formed. If the ordering of attributes and ACs correspond in the AACM and ACAM, then the *attribute interaction matrix (AIM)* is the Boolean resultant matrix from multiplying the AACM and the ACAM:

$$\text{AIM} = \text{AACM} \times \text{ACAM}.$$

The *action cluster interaction matrix (ACIM)* is formed if the order of the matrices in the above multiplication is reversed:

$$\text{ACIM} = \text{ACAM} \times \text{AACM}.$$

The AIM and ACIM are important analysis tools that are explained in greater detail in Chapter 4, as well as [Nance and Overstreet 1987a; Beams 1988].

3.1.2 Action Cluster Incidence Graph

The *action cluster incidence graph (ACIG)*, which is produced from the ACAG, is a digraph that is useful in depicting a certain type of interaction among action clusters. A directed edge is drawn from AC_i to AC_j, if an action of AC_i has the potential to cause AC_j to occur by making the condition of AC_j true by changing the value of one or more control attributes of AC_j.

The algorithm, taken from [Nance and Overstreet 1987a, b], for producing the ACIG looks at common attributes among ACs. An ACIG for a CS consisting of a set of ACs, $\{ac_1, ac_2, \ldots, ac_n\}$, can be constructed as follows:

(1) For each $1 \le i \le n$, let node i represent ac_i

(2) For each ac_i, partition the attributes into three sets:
T_i = {control attributes that are time-based signals}
C_i = {all other control attributes}

O_i = {output attributes}

(3) For each $1 \leq i \leq n$,
For each $1 \leq j \leq n$,
Construct a solid edge from node i to node j if $O_i \cap C_i \neq \emptyset$
Construct a dashed edge from node i to node j if $O_i \cap T_i \neq \emptyset$
END for each $1 \leq i \leq n$
END for each $1 \leq j \leq n$.

An example of the ACIG for the Single Server Queue Model is shown in Figure 8.

The algorithm above produces a graph that contains many potential edges; however, some edges from node i to node j reveal that AC_i may never cause AC_j to occur. These edges, called infeasible edges, are removed from the ACIG to produce a *simplified ACIG*. Chapter 5 describes the method of determining infeasible edges and producing the simplified ACIG. Figure 9 shows the simplified ACIG for the Single Server Queue Model.

3.1.3 Categories of Diagnostic Assistance

Nance and Overstreet [1986, 1987a, b] describe three categories of diagnostic assistance used in automatic analysis of the CS for a discrete-event simulation model: (1) analytical, (2) comparative, and (3) informative. Table 5, reprinted from [Nance and Overstreet 1987a], classifies the diagnostic techniques by category and shows which graph forms the basis of its diagnosis. Analytical diagnostics, which are the easiest to automate, determine if a model representation possesses a certain property. Comparative diagnostics provide a relative measure of a representation characteristic that is compared with other models to interpret the results successfully. Informative diagnostics, which are the most knowledge intensive and include model derivations or extracts that could assist

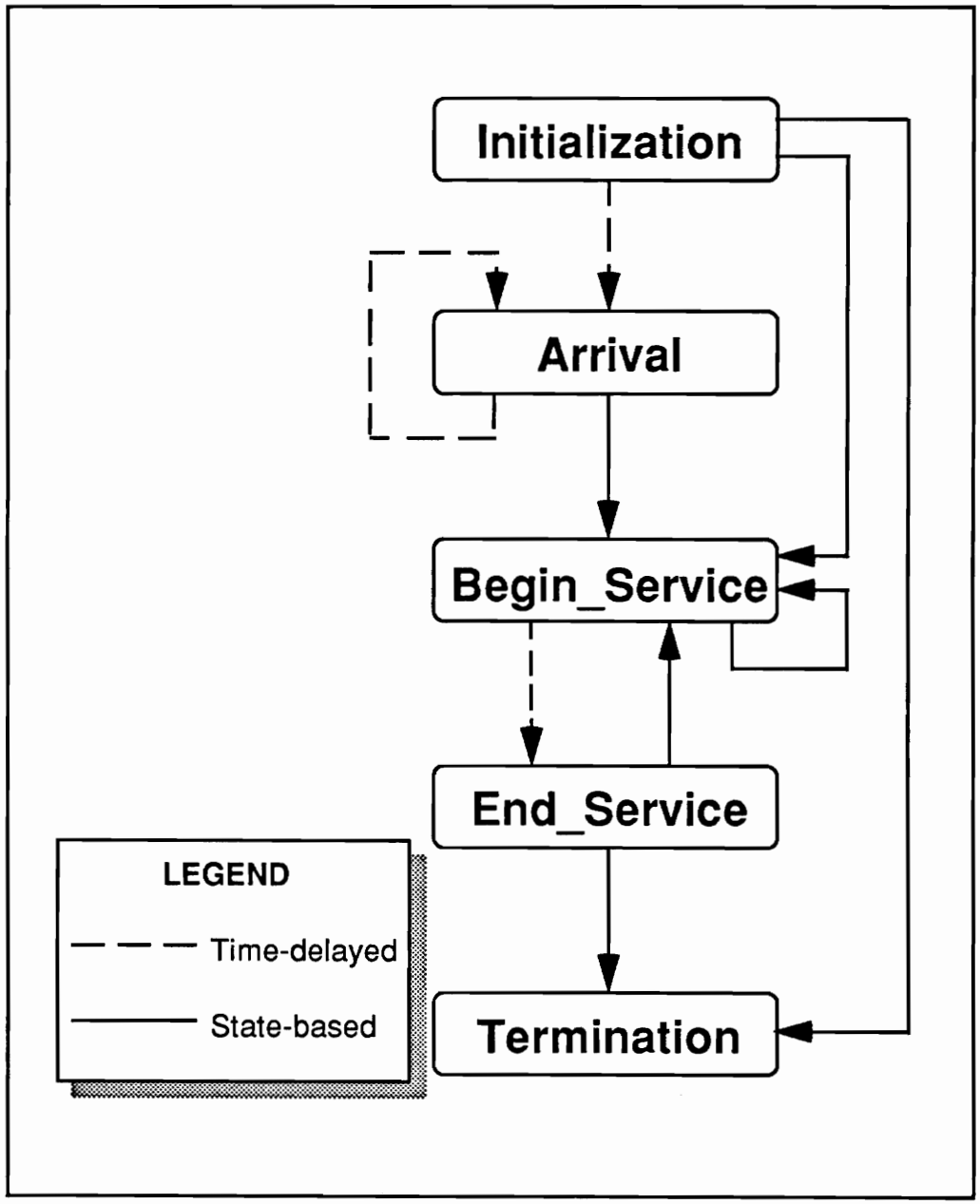


Figure 8. Unsimplified Single Server Queue Action Cluster Incidence Graph

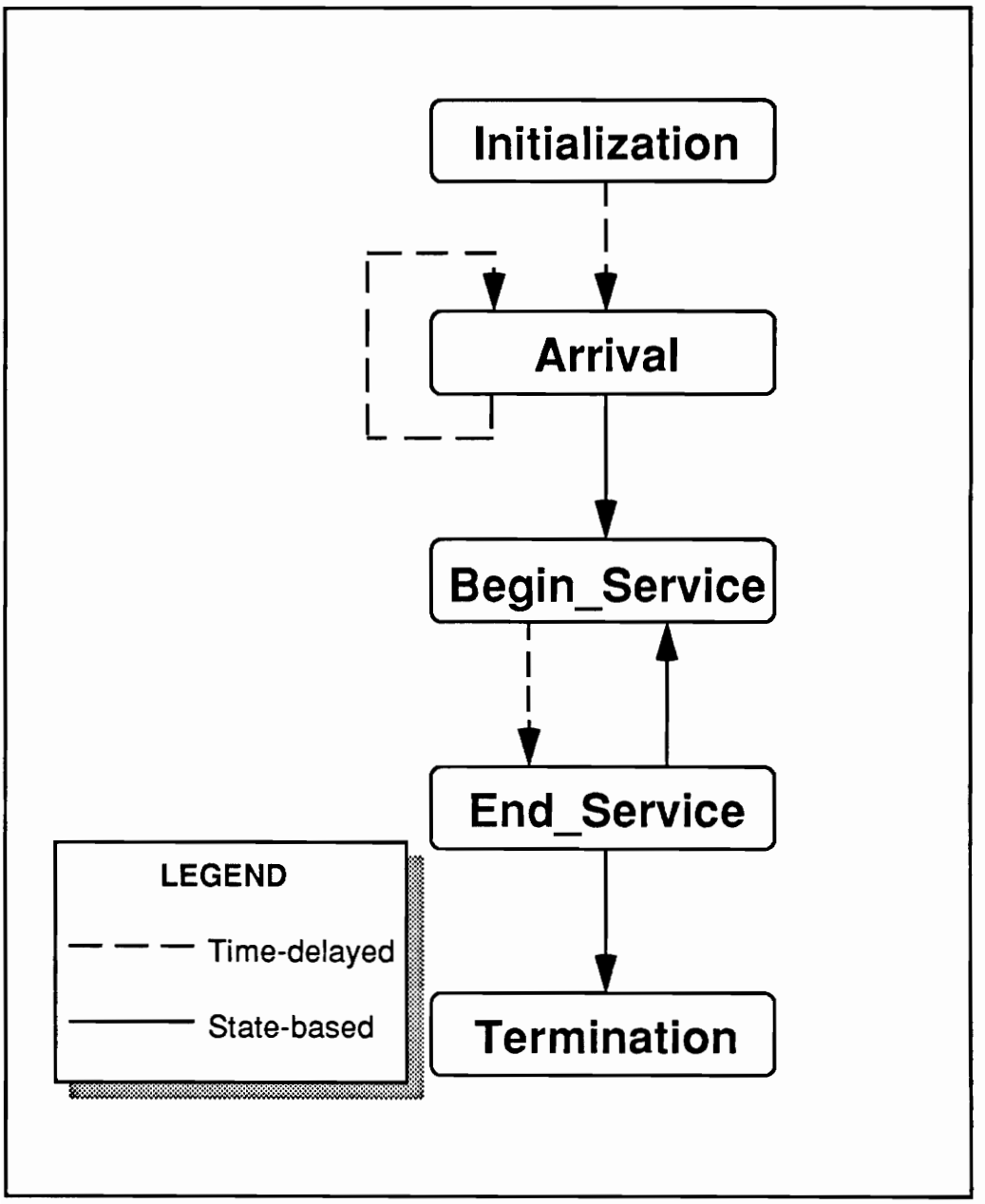


Figure 9. Simplified Single Server Queue Action Cluster Incidence Graph

Table 5. Categorized Summary of Diagnostic Assistance
[Nance and Overstreet 1986, p. 27]

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 of a model representation.	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 Graph (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 termination 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 (originates 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 overhead) criteria.	ACIG
3) Informative: Characteristics 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

in verification or correctness of the model, must rely on the modeler's ability to recognize something instructive.

3.2 Previous Analyzer Prototype

In 1985, Moose [Moose and Nance 1987, 1989] developed an early Model Analyzer prototype for the SMDE Project that uses the graph-based analysis approach of Nance and Overstreet [1986, 1987a, b]. Moose developed the analyzer prototype under the VAX 11/785 and also used an absolute device, a Ramtek 6221 color graphics terminal, to display graphical output. The prototype provides diagnostic analysis of non-executable model representations, namely the Condition Specification, and also provides automated and semi-automated support in the model development process.

The use of the UNIX operating system and Lex and Yacc are two important design decisions in this prototype. Lex and Yacc, described in Section 4.4.2, are meta-tools used to parse the input model representation. Moose treats the CS and its components and graphical representations as abstract data types. Lists, queues, and hash tables are also used as low level data structures.

As outlined in [Moose and Nance 1987, 1989], the prototype adheres to the following steps:

(1) Takes as input a Condition Specification of a model.

(2) Parses the representation and halts if an error is detected.

(3) Compiles diagnostic information about the model representation.

(4) Allows the modeler, through a text-based selection menu, to view components of the model representation and pieces of the diagnostic information.

(5) Produces and displays (if a Ramtek device is available) the ACIG.

The input Condition Specification is a list of condition-action pairs that follows a Pascal-like syntax. Error detection during the parsing phase does not allow provisions for error recovery or reporting and lacks some error detection capabilities, such as type checking. If an error is found, the program halts with only the message "syntax error". Also during the parsing phase, the entire CS representation is stored in a composite data structure. Objects, CAPs, and the set of input, output, and control attributes for each AC are saved. Three pieces of diagnostic information are also compiled for each attribute: whether an attribute is (1) referenced, (2) used as a control or input attribute, and (3) initialized.

After completing a successful parsing of the CS, the modeler enters an interactive session. By typing in a number corresponding to a menu item of a list, the modeler can view various components of a model and the previously described attribute-based diagnostic information. Some additional typed input, such as AC or attribute names, also may be needed. The output is listed in a text-based tabular form.

After the interactive session, the ACIG is constructed and displayed in two forms. First the adjacency matrix of the ACIG is displayed on the terminal, and then the traditional graphical representation of the ACIG is displayed on a separate Ramtek output device.

The prior prototype is limited in its functionality, although it may be easily expanded. Of the fourteen diagnostic techniques listed in Table 5, only one-half of them have been implemented in the prototype. The properties of attribute utilization, attribute initialization, and attribute classification are completed, but the graph-based diagnostics of connectedness, accessibility, and out-completeness are in various stages of

development. The complexity measure, which implements Wallace's Control and Transformational (CAT) metric described in Section 4.5.3.2.3, is also not completed.

3.3 Analyzer Prototype Objectives

The prior prototype provides important knowledge about performing model analysis and interacting with other tools in the SMDE. This research applies this gained knowledge to build a Model Analyzer prototype that has expanded capabilities over the prior prototype. The objectives of the new prototype are to:

(1) provide automated and semi-automated support to the modeler,

(2) perform diagnostic analysis on non-executable model specifications, called Condition Specifications,

(3) perform analytical, comparative, and informative diagnosis on both static and dynamic aspects of the model,

(4) use graph-based diagnostic techniques,

(5) provide the user with graphical representations of the model,

(6) interact with the user through a graphical user interface, and

(7) use a relational database to store the CS and other information about the model.

The sixth and seventh objectives have never been included in previous analyzer prototypes. The new prototype also provides more than one graphical representation of the model and includes more diagnostic assistance than ever before.

4. Model Analyzer Design

This chapter describes in detail the design, implementation, and capabilities of the Model Analyzer prototype (from now on simply called the Model Analyzer). The Model Analyzer, its documentation, and related files consist of over 16,500 lines of C and EQUEL-C code. The architecture for the Model Analyzer, and the SMDE, is a SUN-3/160 color computer workstation running a Berkeley 4.2 based UNIX operating system. The SUN workstation also includes a 19-inch color monitor with a resolution of 1152×900 pixels, a 3-button mouse, and a keyboard.

The Model Analyzer can be invoked two different ways: (1) from the SMDE or (2) from the SunView command line. It can be started from the SMDE by selecting the "Model Analyzer" button in the top panel of the SMDE, shown in Figure 10. Figure 11 shows how the screen looks after the "Model Analyzer" button is selected. The Model Analyzer operates within the SMDE providing the user has the proper priority and clearance to access the database and other files. Currently, the tools in the SMDE are not linked together and function individually. The goal of the project is to have a working integrated environment. In this integrated environment, the Model Generator produces the Condition Specification that is used as input to the Model Analyzer, and the Assistance Manager, an on-line help facility, is invoked from within the Model Analyzer (or any other tool).

Since the SMDE is currently not integrated, the second method of invoking the Model Analyzer is preferred. The prototype can be activated within SunView by typing

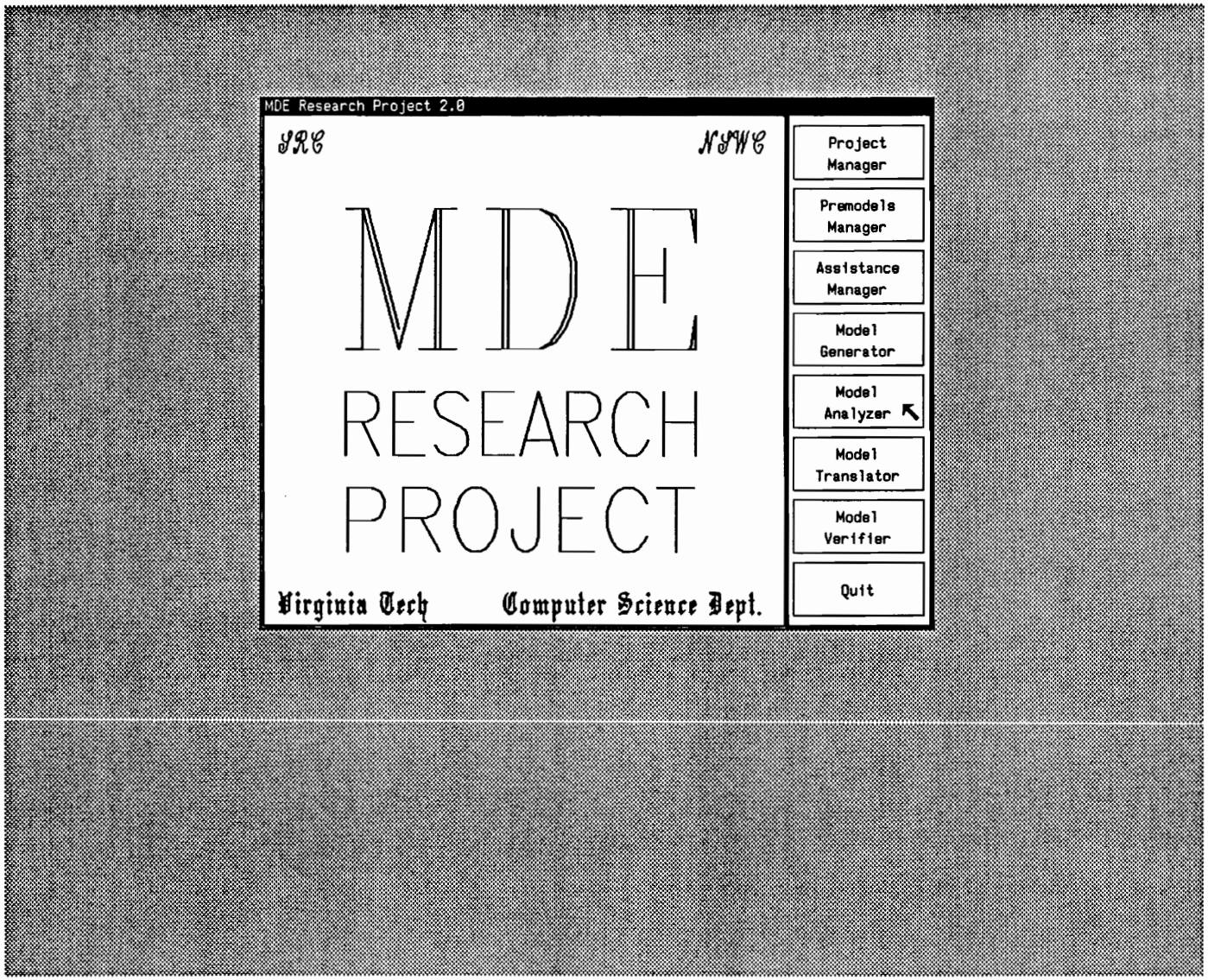


Figure 10. The Simulation Model Development Environment

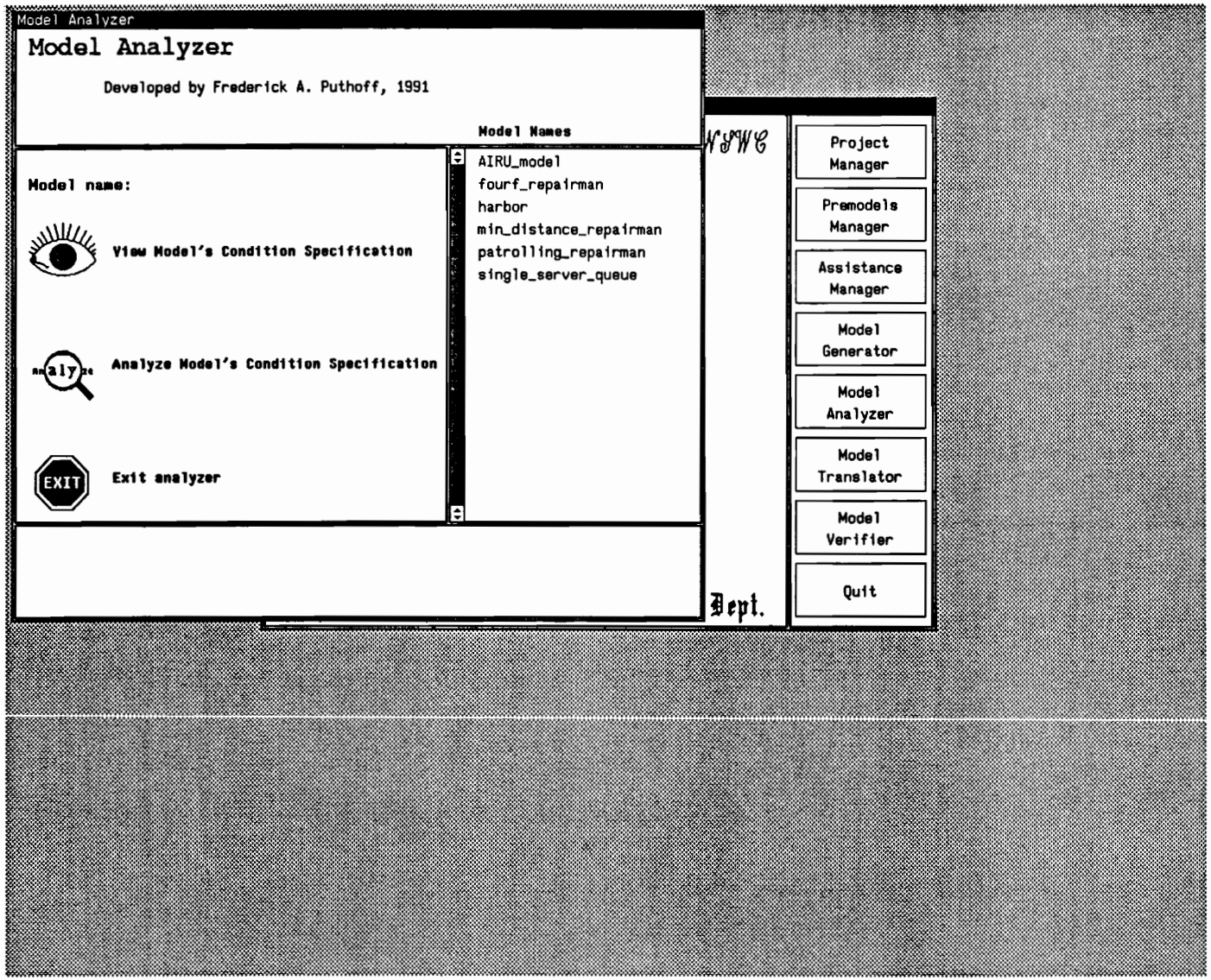


Figure 11. The Model Analyzer in the Simulation Model Development Environment

the word "analyzer" on the command line while in the directory "usr/puthoff/analyzer". Figure 12 shows the top level of the Model Analyzer after activation.

Section 4.1 describes the design of the Model Analyzer interface. The functions at the highest level of the Analyzer are explained in Section 4.2. The construction of the database is given in Section 4.3. Section 4.4 discusses how the Model Analyzer parses the input Condition Specifications, and the graphical analysis techniques are explained in Section 4.5.

4.1 Human-Computer Interface Design

Since the Model Analyzer is a prototype and not intended to be a marketable product, many user interface issues are not of concern in this prototype. However, to enhance the usability of the analysis techniques, the interface must not be a hindrance. The design principles outlined in Section 2.3 are followed in designing the human-computer interface for the Model Analyzer. Section 4.1.1 describes the User Action Notation as an important interface specification tool. Details of the graphical user interface used in the design of the Model Analyzer interface are listed in Section 4.1.2.

4.1.1 Interface Specification Using User Action Notation

User Action Notation (UAN) [Hartson et al. 1990] is a tool-supported behavioral technique for representing the user interface of interactive systems. Initially developed for communicating behavioral descriptions of interface designs to implementers for construction and to evaluators for a pre-prototype view of the design, UAN provides a behavioral description of the actions a user performs to accomplish a task.

The design of a user interface is specified as a set of quasihierarchical tasks that are asynchronous. Lower level tasks are comprised of user actions, and higher level

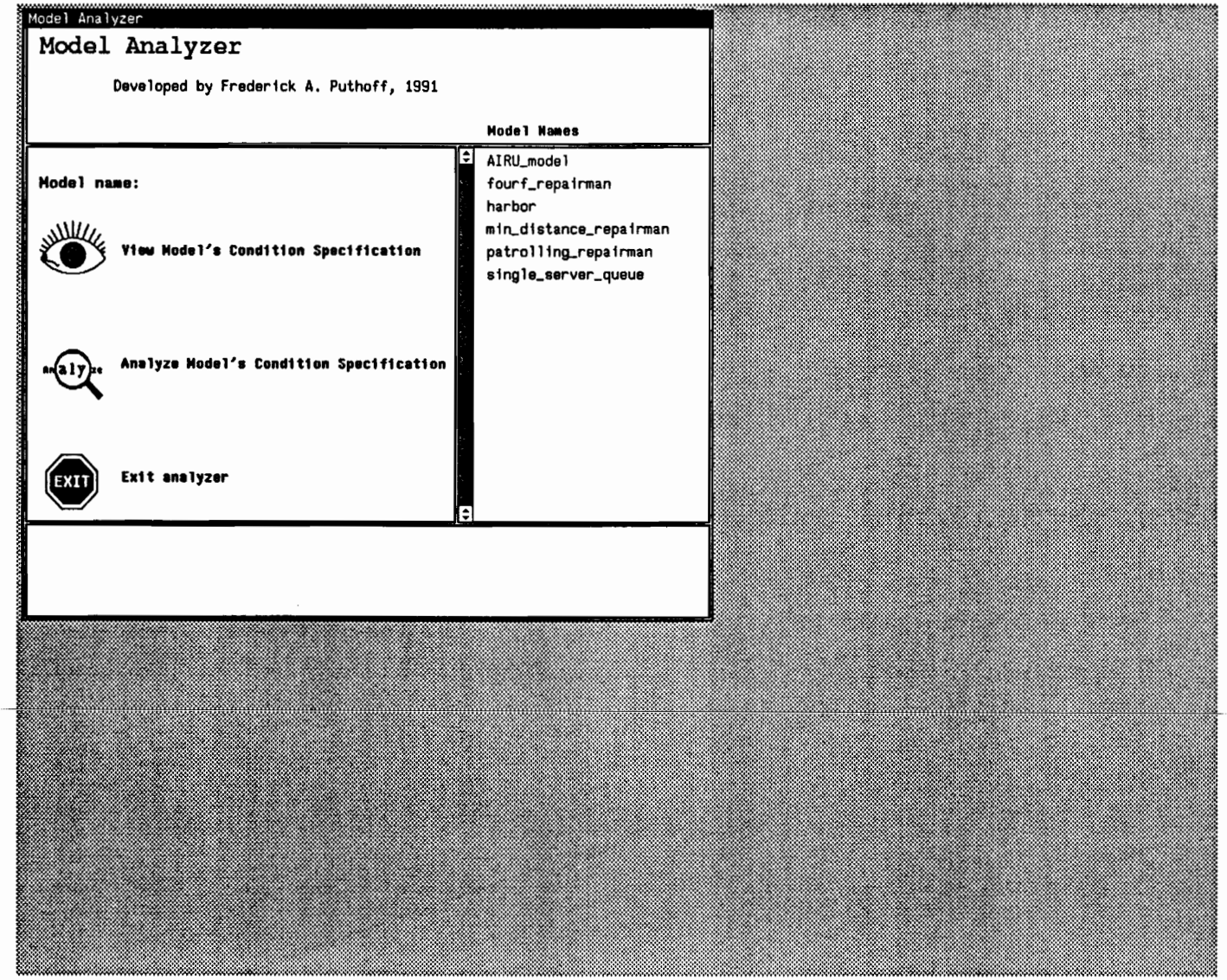


Figure 12. The Top Level of the Model Analyzer

tasks are composed of subtasks. Each task is then decomposed as a series of actions, which describe what the user physically does while using the interface. For each action, interface feedback occurs as the response to the user actions. Besides feedback, state information for the interface and the connections to computation can be changed.

A UAN diagram for a task has four columns: (1) user actions, (2) interface feedback, (3) interface state, and (4) connection to computation. To interpret the task, the table is read from left to right, top to bottom. An example of a UAN diagram for the task of deleting a file from a Macintosh desk top computer is shown in Figure 13.

The UAN symbols are selected with three requirements in mind: (1) usage separate from definition, (2) typeable from a standard keyboard, and (3) mnemonically meaningful. The symbols should convey a visual meaning. For example, the symbol ~ represents movement, and *[X]* conveys the idea of a box around *X*. Table 6 lists a summary of the UAN symbols and their meanings. Relying on mnemonicity and simplicity, UAN is easy to learn and write, yet very precise and clear.

Although UAN is a very useful notation for describing the design of a user interface, it should not be the only representational technique employed. Techniques, such as scenarios, task transition diagrams, and discussion sheets, should accompany UAN diagrams to design user interfaces successfully. The UAN diagrams for the user interface of the Model Analyzer are listed in Appendix A.

4.1.2 Graphical Interface Design Details

The Model Analyzer is developed using SunView, (Sun Visual/Integrated Environment for Workstations), which is a user-interface toolkit to support interactive, graphics-based applications running under windows [Sun Microsystems 1988b, c]. SunView contains four types of windows:

TASK: Delete a Macintosh file			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[file_icon] M∨	file_icon-!: file_icon!, ∀ file_icon'!: file_icon'-!	selected = file	
~[x,y]*	outline(file_icon) > ~		
~[trash_icon]	outline(file_icon) > ~, trash_icon!		
M∧	erase(file_icon), trash_icon!!	selected = null	mark file for deletion

Figure 13. UAN Example — Deleting a Macintosh File [Hartson et al. 1990, p. 190]

Table 6. Summary of Some Useful UAN Symbols [Hartson et al. 1990, p. 194]

Action	Meaning
~	move the cursor
[X]	the context of object X, the "handle" by which X is manipulated
~[X]	move cursor into context of object X
~[*x*,*y*]	move the cursor to (arbitrary) point *x*,*y* outside any object
~[*x*,*y* in A]	move the cursor to (arbitrary) point within object A
~[X in Y]	move to object X within object Y (e.g., [OK_icon in dialogue_box])
[X]~	move cursor out of context of object X
∨	depress
∧	release
X∨	depress button, key, or switch called X
X∧	release button, key, or switch X
X∨∧	idiom for clicking button, key, or switch X
X"*abc*"	enter literal string, *abc*, via device X
X (*xyz*)	enter value for variable *xyz* via device X
( )	grouping mechanism
*	iterative closure, task is performed zero or more times
+	task is performed one or more times
{ }	enclosed task is optional (performed zero or one time)
A B	sequence; perform A, then B (same if A and B are on separate, but adjacent, lines)
OR	disjunction, choice of tasks (used to show alternative ways to perform a task)
&	order independence; connected tasks must all be performed, but relative order is immaterial
⇔	interleavability; performance of connected tasks can be interleaved in time
‖	concurrency; connected tasks can be performed simultaneously
;	task interrupt symbol; used to indicate that user may interrupt the current task at this point (the effect of this interrupt is specified as well, otherwise it is undefined, i.e., as though the user never performed the previous actions)
∀	for all
:	separator between condition and action or feedback
Feedback	**Meaning**
!	highlight object
-!	dehighlight object
!!	same as !, but use an alternative highlight
!-!	blink highlight
(!-!)^n	blink highlight *n* times
@ *x*,*y*	at point *x*,*y*
@ X	at object X
display (X)	display object X
erase (X)	erase object X
X > ~	object X follows (is dragged by) cursor
X >> ~	object X is rubber-banded as it follows cursor
outline (X)	outline of object X

(1) *canvases* on which programs can draw,

(2) *text subwindows* with built in editing capabilities,

(3) *panels* containing items such as buttons, choice items, and analog sliders, and

(4) *tty subwindows* in which programs can be run.

The windows are arranged as subwindows within frames, which are windows themselves, and the windows may be overlapped, resized, "closed", or collapsed into an icon, and "reopened". Icons provide a visual feedback and are attractive to users. Scrollbars also may be attached to the first three types of windows. Menus and alerts also may "pop-up" anywhere on the screen.

The Model Analyzer user interface is graphical, which is obvious from the diagrams presented in this chapter. It contains canvases, panels, text subwindows, icons, buttons, multiple choice items, scrollbars, menus, and alerts. The Model Analyzer also has a "point and click" interface. All user input is done with a click of a mouse button, eliminating the need for typing and emphasizing user convenience. Typing is required in the Model Analyzer only for editing within the Model Viewer, described in Section 4.2.2.

4.2 Model Analyzer Top Level

When the Model Analyzer is invoked, the highest level, shown in Figure 12, is displayed. The modeler must select a model from the list by selecting a model name with the mouse. Once a model is selected, its name appears to the right of the words "Model name:". If the modeler attempts to view or analyze the CS of a model before selecting a model, an alert pops up on the screen as shown in Figure 14.

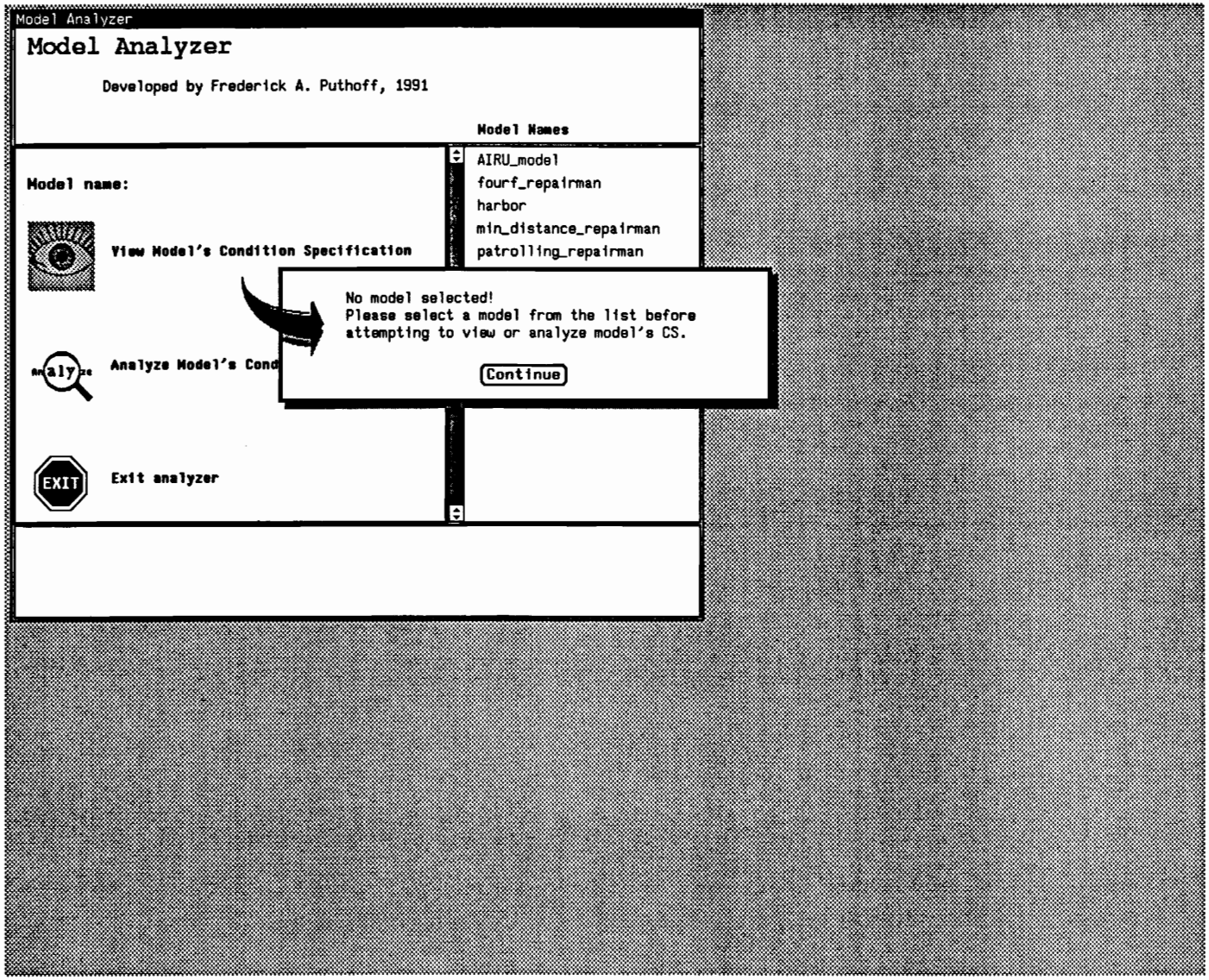


Figure 14. No Model Selected Alert in Top Level of Model Analyzer

The top level of the Model Analyzer can perform three tasks: (1) exit the analyzer, (2) view the CS of a model, and (3) analyze the CS of a model. Each of these three tasks is described in the following three sections.

4.2.1 Exiting the Model Analyzer

The modeler can exit the Model Analyzer by selecting the iconic button that looks like a stop sign, shown in Figure 12. After selecting the button, an alert message appears requesting the modeler to quit.

4.2.2 Viewing the Condition Specification of a Model

To view the CS of a model, the modeler selects the iconic button that resembles a huge eyeball after choosing a model from the list. Figure 15 shows the screen after the modeler selects the "single_server_queue" (Single Server Queue) model and depresses the view button. The display panel, called the Model Viewer, shows the input to the Model Analyzer, which is the CS of the selected model, within a scrollable text subwindow. The CS file is currently a read-only file. The "Edit" button must be pressed before the modeler can attempt to edit and change the CS. The "Quit/No Save" button returns the modeler to the top level and ignores any changes made in the CS; whereas, the "Return/Save" button returns to the top level and saves the editing changes, if any.

When the SMDE is totally integrated, editing the model CS within the final Model Analyzer is prohibited. If the CS needs to be altered, changes should take place with the help of the Model Generator. Since the SMDE is currently not integrated, allowing the modeler to edit the CS within the Model Analyzer is simply a needed convenience.

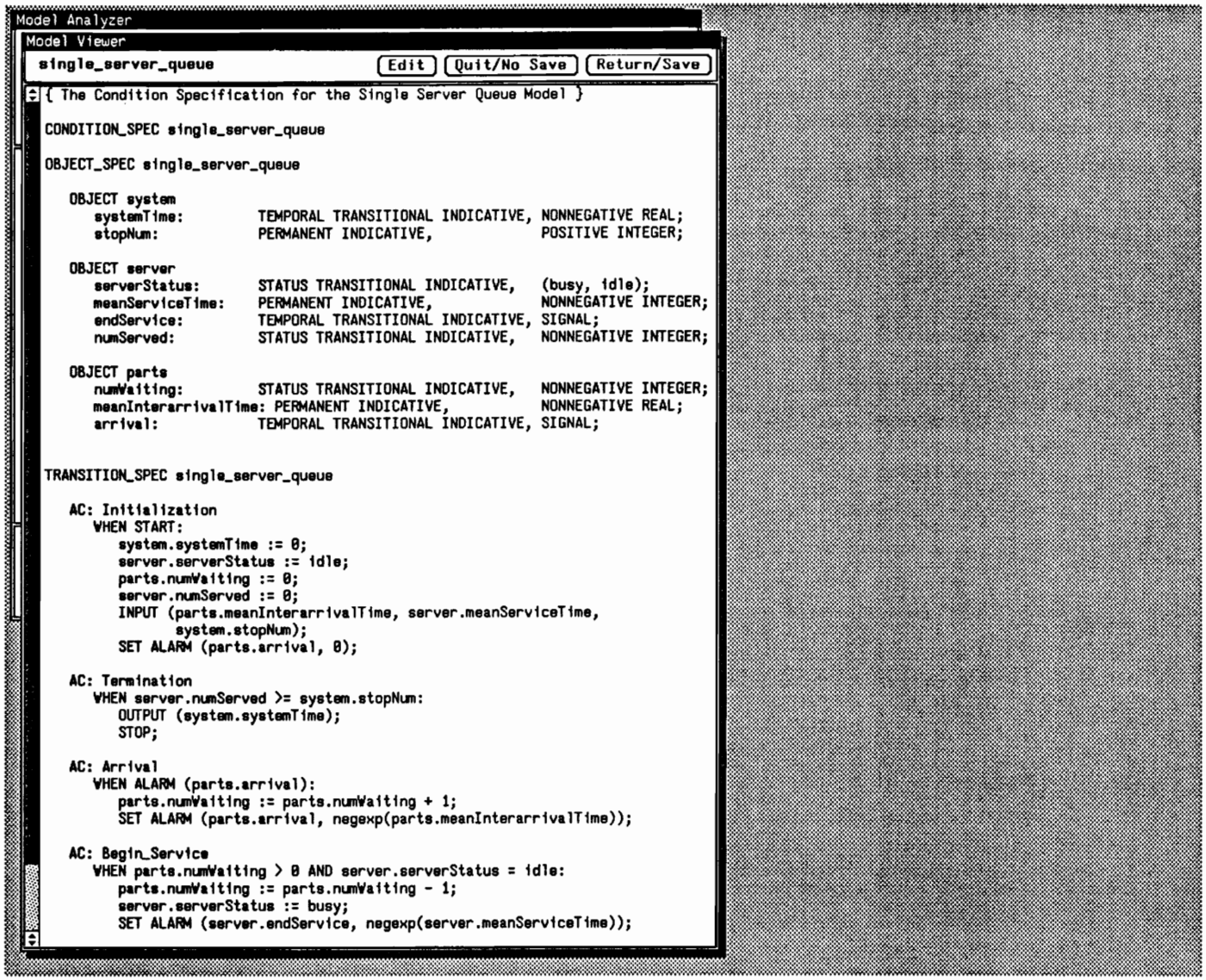


Figure 15. The Model Viewer within the Model Analyzer

4.2.3 Analyzing the Condition Specification of a Model

To analyze the CS of a model, the modeler must first select the iconic button that looks like a magnifying glass after choosing a model from the list. Before a model can be analyzed, the Model Analyzer performs the following steps:

(1) changes the cursor to an hourglass and prints a wait message on the screen, shown in Figure 16,

(2) creates the relational database and its relations,

(3) parses the CS and stores the information into the database,

(4) checks if the parse is successful (An error in parsing the CS causes an error message to be displayed and the analysis preparation stops),

(5) creates a numbered list of action clusters and attributes,

(6) builds the ACAG and ACIG,

(7) builds the AIM and ACIM,

(8) returns the cursor to an arrow, and

(9) displays the Analyze Model panel.

The analysis preparation steps are time consuming and will take from two to five minutes to complete for complicated models. After the Analyze Model panel is displayed, the CS of the model is ready to be analyzed.

4.3 Database Construction

The Model Analyzer uses INGRES [Stonebraker et al. 1976; Sun Microsystems 1985, 1987], a powerful relational database management system (DBMS), to store and retrieve CS and analysis information related to the current model. INGRES, which stores

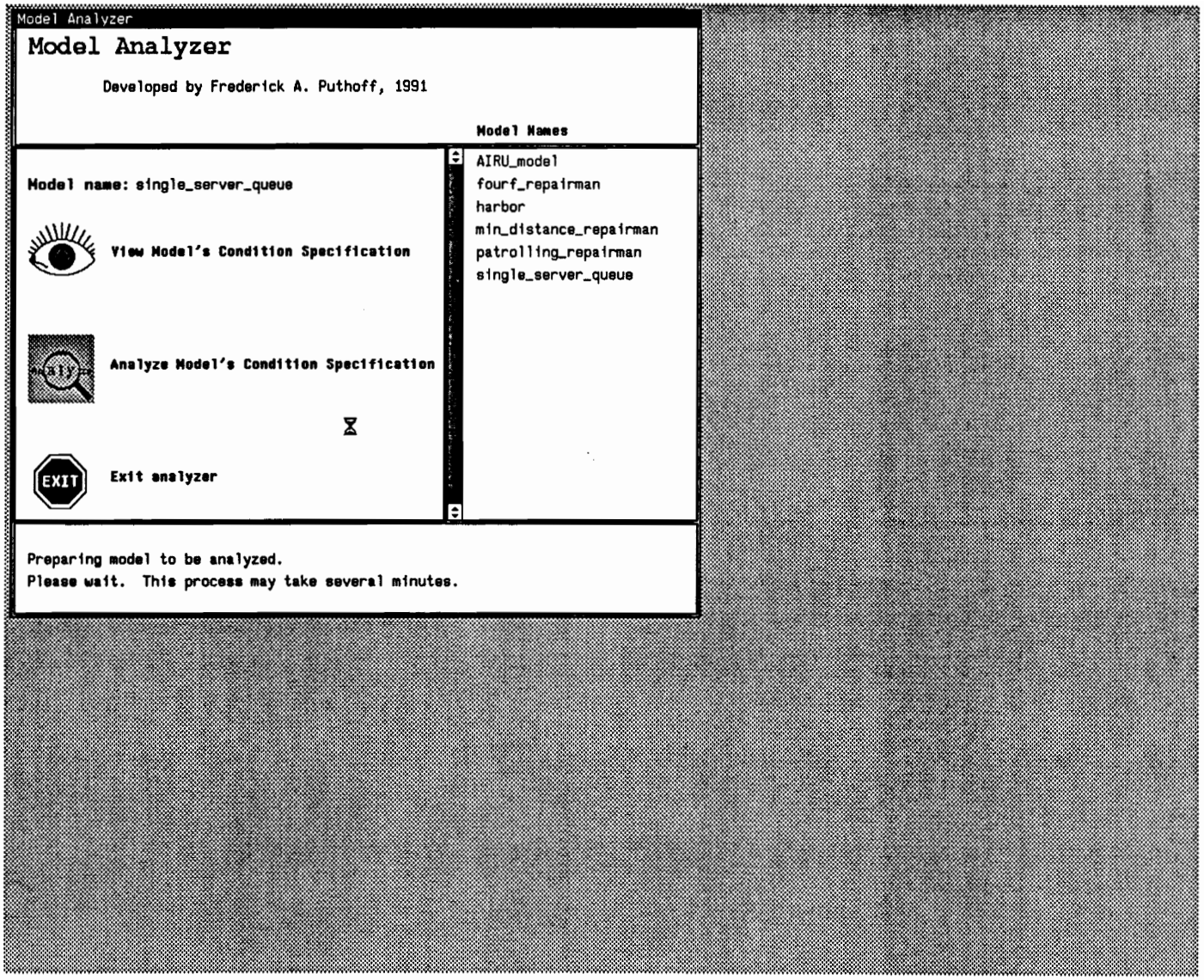


Figure 16. Preparing to Analyze the Model

its data in tables called relations, includes many subsystems that help the user to enter, query, update, and analyze data.

EQUEL/C (Embedded QUEry Language in C) statements access the INGRES database from within the general purpose programming language C. EQUEL/C statements are simple expressions used to retrieve, insert, delete, and update data stored in the database.

The Model Analyzer creates a database called *madbase*, which simply stands for **M**odel **A**nalyzer **d**ata**base**. *Madbase* consists of approximately twenty relations that store the CS and analysis information in an understandable format for the Model Analyzer. Some relations are used to store parts of the CS, such as its name, objects, attributes, action clusters, conditions, actions, functions, and local variables. Other relations provide important analysis information, such as the ACAG, unsimplified ACIG, simplified ACIG, and the classification of attributes within each action cluster. These relations are described in detail in the program documentation for *initdb* within the file *database.qc*.

4.4 Parsing the Condition Specification

In the SMDE, the Model Generator generates a file that contains the Condition Specification as a series of action clusters (not CAPs) for a model. The CS is the input used by the Model Analyzer to perform its analysis functions. Section 4.4.1 defines the structure or syntax of the CS. Before the CS can be analyzed, the CS must be parsed and its information put into the relational database. Lex and Yacc provide this capability to take a CS representation and load the database with the correct information. Section 4.4.2 describes Lex, Yacc, and the parsing process.

4.4.1 Condition Specification Syntax

The syntax of the Condition Specification is the same as described in Section 2.4 and Table 2 plus a few additions. The formal notation for the CS syntax is defined in Appendix B, and an informal description follows.

In addition to the primitives outlined in Table 2, three complex primitives are described in Table 7: (1) FOR SOME, (2) FOR ALL, and (3) FOR loop. The FOR SOME and FOR ALL primitives act as Boolean expressions within a state-based condition; whereas, the FOR loop is a compound action. Each FOR loop is composed of an indexing attribute, a range, and a series of one or more actions. The series of actions is repeated for every value within the range of the index.

Another important feature of the CS is that each attribute uses the dot notation to represent object ownership. Outside the object specification, each attribute is written in the following form:

<object_type>.<attribute_name>

This dot notation allows the modeler to know instantly the relation of each object to each attribute and permits the same attribute name to be associated with different objects.

The CS also provides a mechanism for objects and attributes to have multiple instances. In the object specification, each object may be followed by a range in square brackets. This allows the modeler to define several objects with identical attributes. An array of attributes also may be represented by an attribute followed by a range of integers in square brackets. Even enumerated values may be shown as an array of values using this indexing approach. When used in the transition specification, every indexed object, attribute, or enumerated value must be immediately followed by a valid index in square brackets that is within a predefined range.

Table 7. Syntax and Function of Additional Condition Specification Primitives

NAME	SYNTAX	FUNCTION
For All expression	FOR ALL <variable_spec> : <attribute_exp> .. <attribute_exp>, <Boolean_exp>	Boolean range expression
For Some expression	FOR SOME <variable_spec> : <attribute_exp> .. <attribute_exp>, <Boolean_exp>	Boolean range expression
For Loop	FOR <variable_spec> := <attribute_exp> TO <attribute_exp> DO <action> [;<action>]*; END FOR	Provide an iterative loop action

Each attribute in the object specification can have a CS typing of integer, real, Boolean, character, string, enumerated, or signal. The CS also provides the Conical Methodology (CM) Type [Nance 1981a, 1987]. Five CM types are valid in the CS syntax: (1) permanent indicative, (2) status transitional, (3) temporal transitional, (4) hierarchical relational, and (5) coordinate relational. An explanation of the Conical Methodology attribute typing is described in [Nance 1981a, 1987].

Examples of Condition Specifications used in the Model Analyzer are shown in Appendix C. All keywords of the CS are reserved and may not be used as objects, attributes, enumerated values, functions, or local variables. A general outline of the CS is given below.

The CS name is supplied after the keyword CONDITION_SPEC. Next, the same CS name is listed after the keyword OBJECT_SPEC. In the object specification, object definitions consist of the keyword OBJECT followed by the object name and a list of attribute definitions. Each attribute definition lists the attribute name followed by its CM and CS type. Besides the CS types listed above, numeric types may be preceded by one of the qualifiers: (1) positive, (2) nonnegative, (3) nonpositive, or (4) negative.

The transition specification, which is introduced by the keyword TRANSITION_SPEC followed by the CS name, consists of a set of action clusters. Each action cluster (AC) is introduced by the keyword AC followed by a colon and the name of the action cluster. After the name of the AC, the condition and associated actions are listed. A function is recognized by any non-keyword identifier that immediately precedes a "(". Local variables are defined as identifiers that have not been defined as objects, attributes, or enumerated values.

The function and report specifications have been included as optional parts of the CS, but not been formally defined. The function specification may be introduced after

the transition specification by the keyword FUNCTION_SPEC and the CS name; however, the CS does not support the syntax for the body of the function specification. The report specification may be defined after the function specification with the keyword REPORT_SPEC followed by the CS name, but the body is also not supported by the current CS syntax.

4.4.2 Parsing the Condition Specification with Lex and Yacc

Lex [Lesk and Schmidt 1983; Aho et al. 1986, p. 105], a lexical analyzer, is a program generator designed for lexical processing of character input streams. Lex recognizes user defined regular expressions in an input stream and partitions the input stream into tokens matching the expressions. A Lex program also includes actions, which are program fragments describing what action the lexical analyzer should take when a pattern is matched.

Yacc [Johnson 1983; Aho et al. 1986, p. 257], yet another compiler-compiler, provides a general tool for describing the input to a computer program. The Yacc user specifies the structure of the input as a series of grammar rules expressed in Backus-Naur Form (BNF). Yacc accepts LALR(1) grammars with disambiguating rules. When a rule is recognized, the associated user defined actions are invoked.

Lex and Yacc together can be used to decompose an input stream into tokens and then assign structure to the resulting pieces. When Yacc needs a token, it activates Lex. Lex begins reading the remaining input, one character at a time, until if finds the longest input string that matches one of the regular expressions. Lex then performs its action, which usually returns control to Yacc; however, if control is not returned to Yacc, Lex continues to accept more input until an action causes control to be returned. This repeated pattern matching until an explicit return allows Lex to process white space and

comments conveniently. The role of Lex and Yacc in the Model Analyzer is discussed in the next section.

4.4.2.1 Parsing Process of the Model Analyzer

In the Model Analyzer, Lex converts the input CS of a model according to the regular expressions listed in Appendix B.1. The tokens are needed by Yacc to parse the CS according to the BNF grammar rules stated in Appendix B.2. Each rule also has associated actions that are performed as the rule is processed. These actions take the parsed input and insert the information into specific relations of the database.

As the CS is parsed, the entire CS is broken into pieces (such as the name of the CS, objects, attributes, action cluster names, conditions, and actions) and inserted into the proper relations of the database. Besides filling the database with pieces of the CS, each attribute within an action cluster is classified as an input, output, or control attribute according to Table 8. This classification is also stored as a relation in the database.

The classification of attributes, precisely defined in Table 8, provides the basis for the graph-based analysis techniques. Only attributes within each expression are classified, and attributes used as indexes of other attributes, objects, or enumerated values are not classified. For example, the following statement

obj1.att1 := obj2[obj3.index].att2 + 1;

would classify *obj1.att1* as an output attribute and *obj2.att2* as an input attribute. The attribute *obj3.index* is not classified since it is an index to *obj2*. The classification of function attributes depends on the role of the function. If the function is contained in the condition of an action cluster, then the function attributes are typed as control attributes. If the function appears on the right hand side of a value change description, or assignment statement, then the function attributes are typed as input attributes.

Table 8. Classification of Attributes within Action Clusters

Syntax of Action Cluster Primitive	Expression	Classification
<attribute_name> := <attribute_expression>	<attribute_name> <attribute_expression>	output input
SET ALARM (<alarm_name>, <alarm_time> [, <parameter_list>])	<alarm_name> <alarm_time> <parameter_list>	output input output
CANCEL ALARM (<alarm_name> [, <parameter_list>])	<alarm_name> <parameter_list>	output output
WHEN ALARM (<alarm_name_exp> [, <parameter_list>])	<alarm_name_exp> <parameter_list>	control control
AFTER ALARM (<alarm_name_exp> AND <Boolean_exp> [, parameter_list>])	<alarm_name_exp> <Boolean_exp> <parameter_list>	control control control
WHEN <Boolean_expression>	<Boolean_expression>	control
INPUT (<attribute_name> [, <attribute_name>]*)	<attribute_name>	output
OUTPUT (<attribute_name> [, <attribute_name>]*)	<attribute_name>	input

Notes: Only attributes within expressions are classified.
Attributes, which are indexes of objects, attributes, or enumerated values, are not classified.
Attributes of functions are classified by the role of the function within the action cluster.

When the parsing is completed, the database contains all the information needed to analyze the CS representation of the model, and the CS input file is never again accessed. The analysis techniques, described in Section 4.5, access the database to receive the needed information to perform their analysis.

4.4.2.2 Error Detection

The Model Analyzer has a limited capability to detect errors in the CS input. If an error is found while parsing the CS, the parsing process is terminated and a message is displayed on the screen, shown in Figure 17. The message communicates the error, approximate location and line number of the error.

Examples of types of errors categorized by the analyzer are: (1) an improper CS name, (2) two objects of the same name, (3) two attributes of the same name, (4) an undefined attribute, (5) two action clusters of the same name, and (6) other syntax errors. The most notable error detection deficiency is the inability to perform type checking. Attributes of different CS types may be intermixed freely without regard for type compatibility, except for the type *signal*.

4.5 After Parsing Operations

A successful parsing of the CS causes the Model Analyzer to perform a few conditioning operations before the analysis can begin. First, the analyzer creates a numbered list of action clusters in alphabetical order, and a numbered list of attributes also in alphabetical order. These lists simplify the process of displaying menus, set a default order for displaying matrices, and simplify certain analysis operations. Secondly, the ACIG relation is produced according to the algorithm listed in Section 3.1.2. Next, the Model Analyzer loads the ACAG relation in the database according to the process

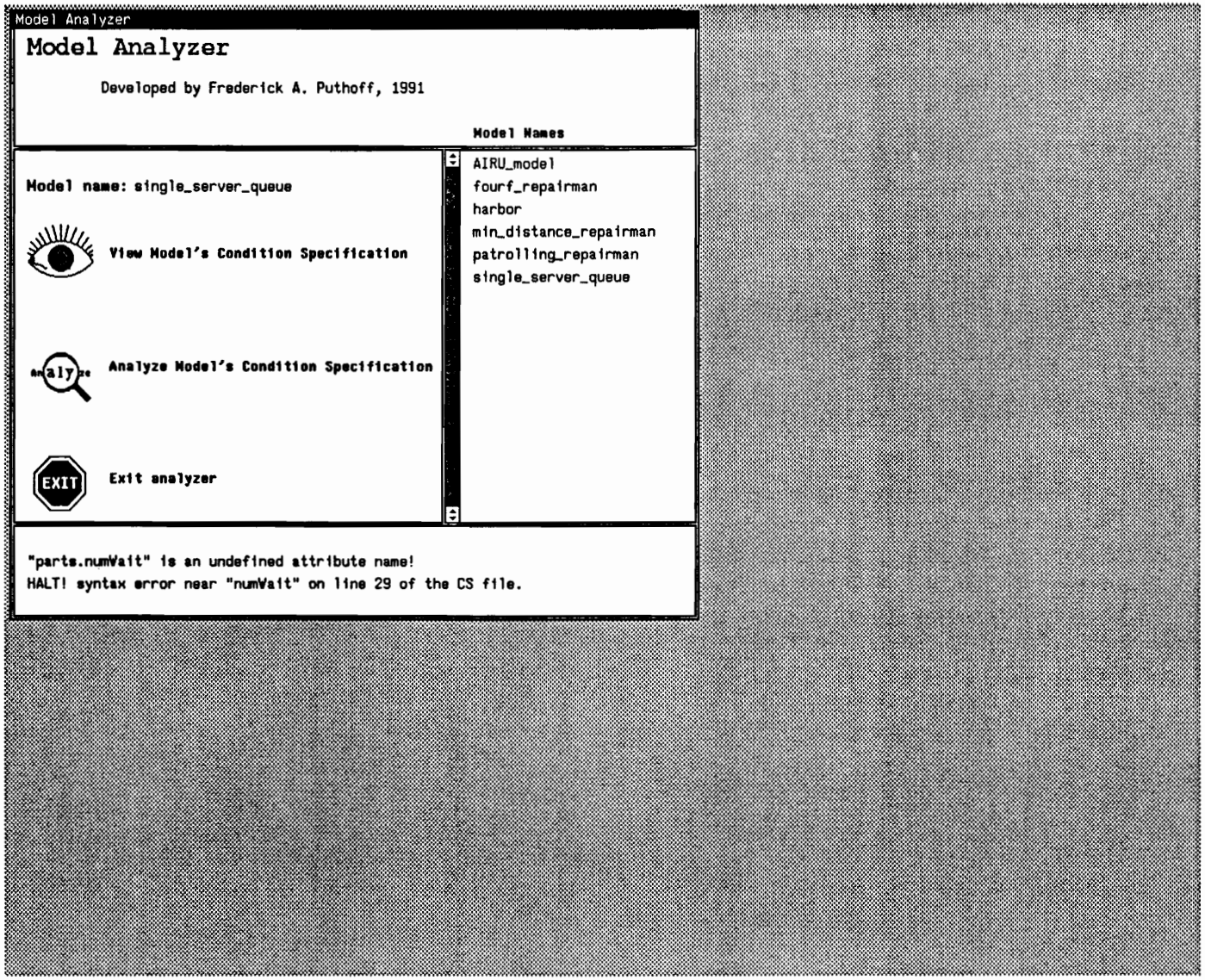


Figure 17. An Error Found While Parsing the Condition Specification

outlined in Section 3.1.1. From the ACAG, the AACM and ACAM are produced and stored in two dimensional arrays. Finally, the AIM and ACIM are created from the AACM and ACAM, as stated in Section 3.1.1, and also stored in two dimensional arrays.

After all the relations and arrays are built as needed, the CS is ready to be analyzed. The analyzer then displays the Analyze Model panel, shown in Figure 18, which lists all the analysis techniques that can be performed upon the model. The panel is broken into three separate sections in which each section represents a different type of analysis. The top section, explained in Section 4.5.1, performs functions that simply list the different parts of the Condition Specification for that model. The lower left section, described in Section 4.5.2, contains four buttons that allow the modeler to view the CS graphically. To the right of the graphical aspects, the diagnostic techniques, defined in Section 4.5.3, are listed.

After finishing analyzing the model, the modeler must select the Return button to return to the top level of the Model Analyzer. When the Return button is selected, an alert message, shown in Figure 19, pops up on the screen asking if the modeler wants to return. This is a safety feature that avoids an accidental return, which means the analyzer must prepare the model to be analyzed again before continuing with the original analysis. At the top level of the Model Analyzer, the modeler may choose to analyze a different model or alter the specification of the current model and then reanalyze the model.

4.5.1 List Procedures

The top section of the Analyze Model panel in the Model Analyzer contains six buttons that are used to list the various components of the model representation and pieces of the diagnostic information. These lists help the modeler determine the correctness of the model specification. By examining these lists, the modeler may see

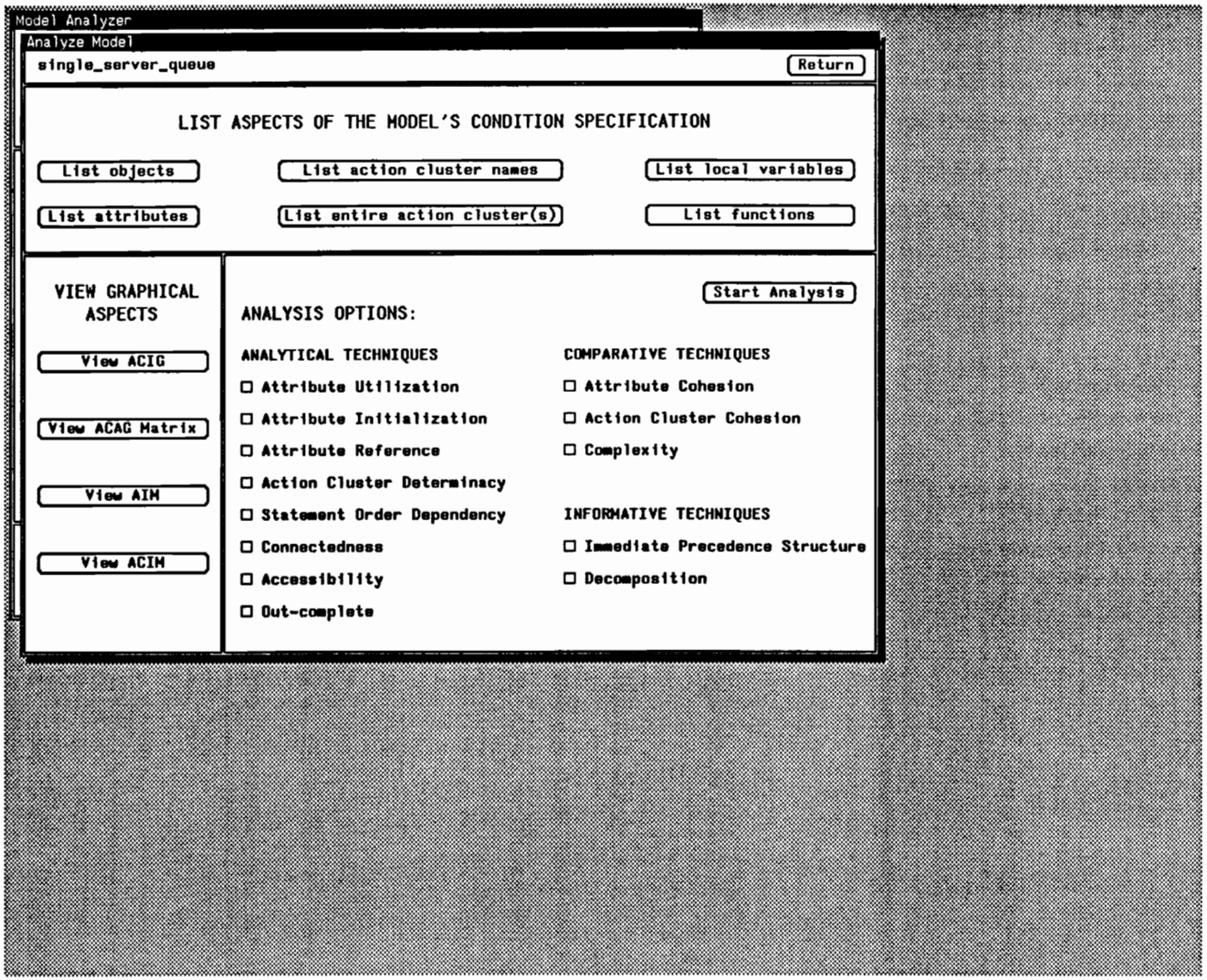


Figure 18. The Analyze Model Panel in the Model Analyzer

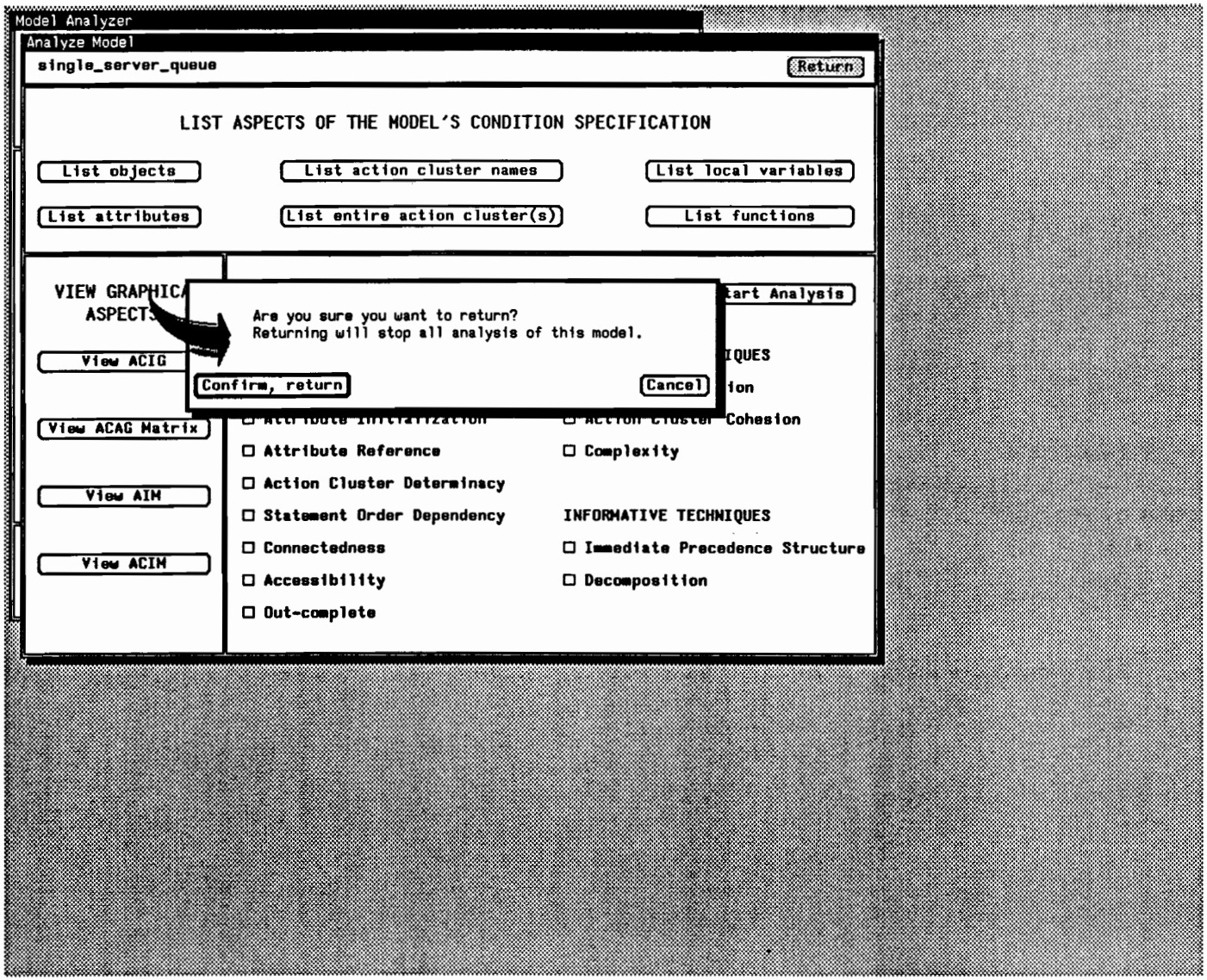


Figure 19. The Alert Message to Stop Analyzing the Model

that part of the specification is incomplete or incorrect. The following six sections describe the capabilities of the six buttons.

4.5.1.1 Objects of the Model

The first button, labeled "List objects", lists the objects of the model. When the "List objects" button is selected, the Model Analyzer displays a panel that lists the objects, shown in Figure 20. The panel also lists the index range, if any, of each object.

4.5.1.2 Attributes of the Model

The "List attributes" button lists the attributes defined in the model along with various diagnostic information about each attribute. This button serves two important purposes. First, it can show how the attributes were defined in the object specification of the model and their parent objects, and secondly, it can show how each attribute is used in the transition specification of the model.

When the "List attributes" button is selected, a menu, shown in Figure 21, appears entitled "ATTRIBUTES:", which has two choices: (1) "Of object(s)" and (2) "Within Action Cluster(s)". Each choice has an arrow after its name signifying this is a *pull-right* menu in which another menu is displayed by following the right pointing arrow of each choice with the mouse.

The first of two choices in the "ATTRIBUTES:" pull-right menu, "Of Object(s)", performs the purpose of listing the definition of the attributes in each object. Following the arrow next to "Of Objects" displays another pull-right menu entitled "ATTRIBUTE TYPE:", which has three choices: (1) "All Types", (2) "Indicative", and (3) "Relational". Following the right pointing arrow of any of these three choices displays the final menu entitled "FOR OBJECT:", which has "All Objects" and each object defined in the model

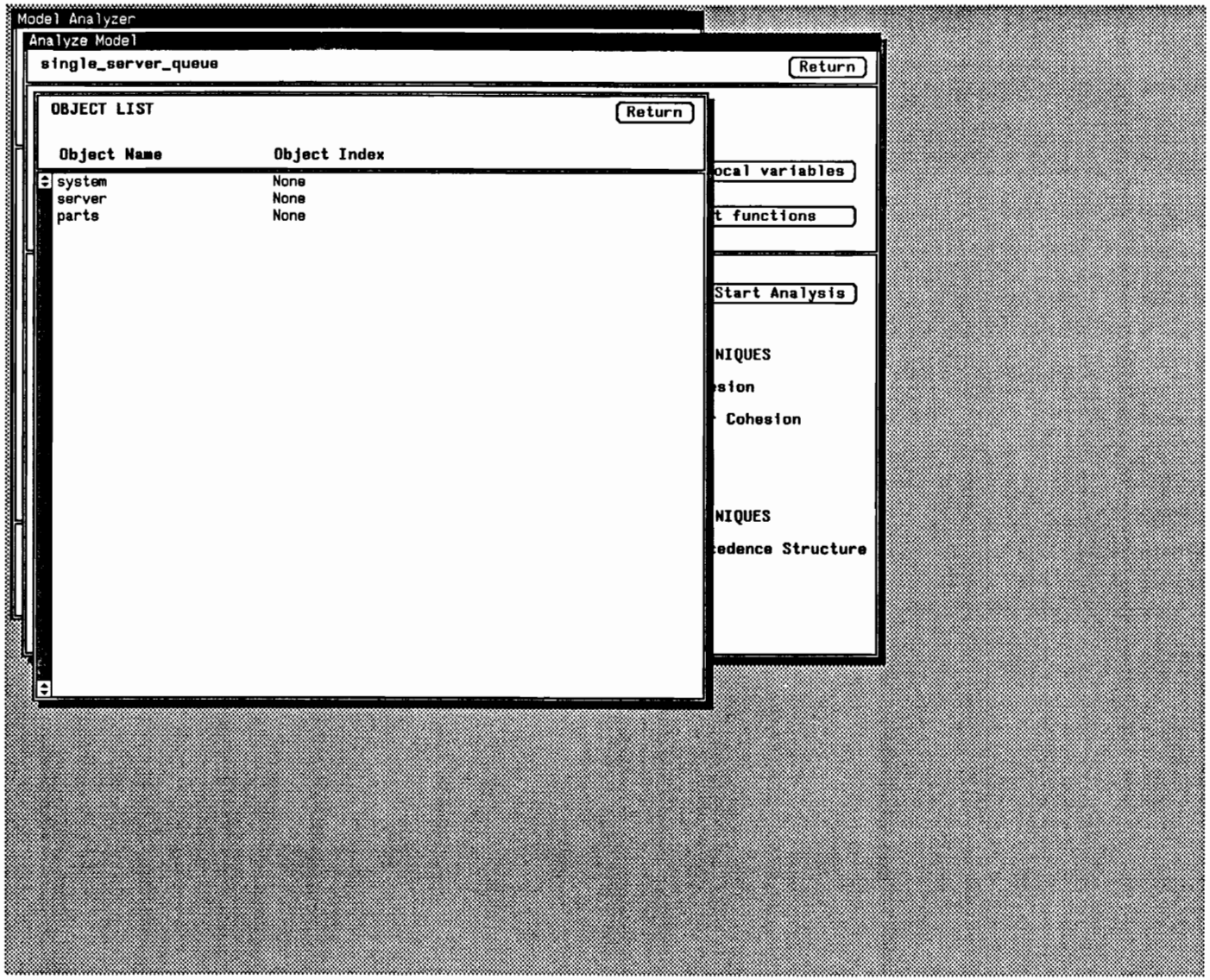


Figure 20. The Object List of the Single Server Queue Model

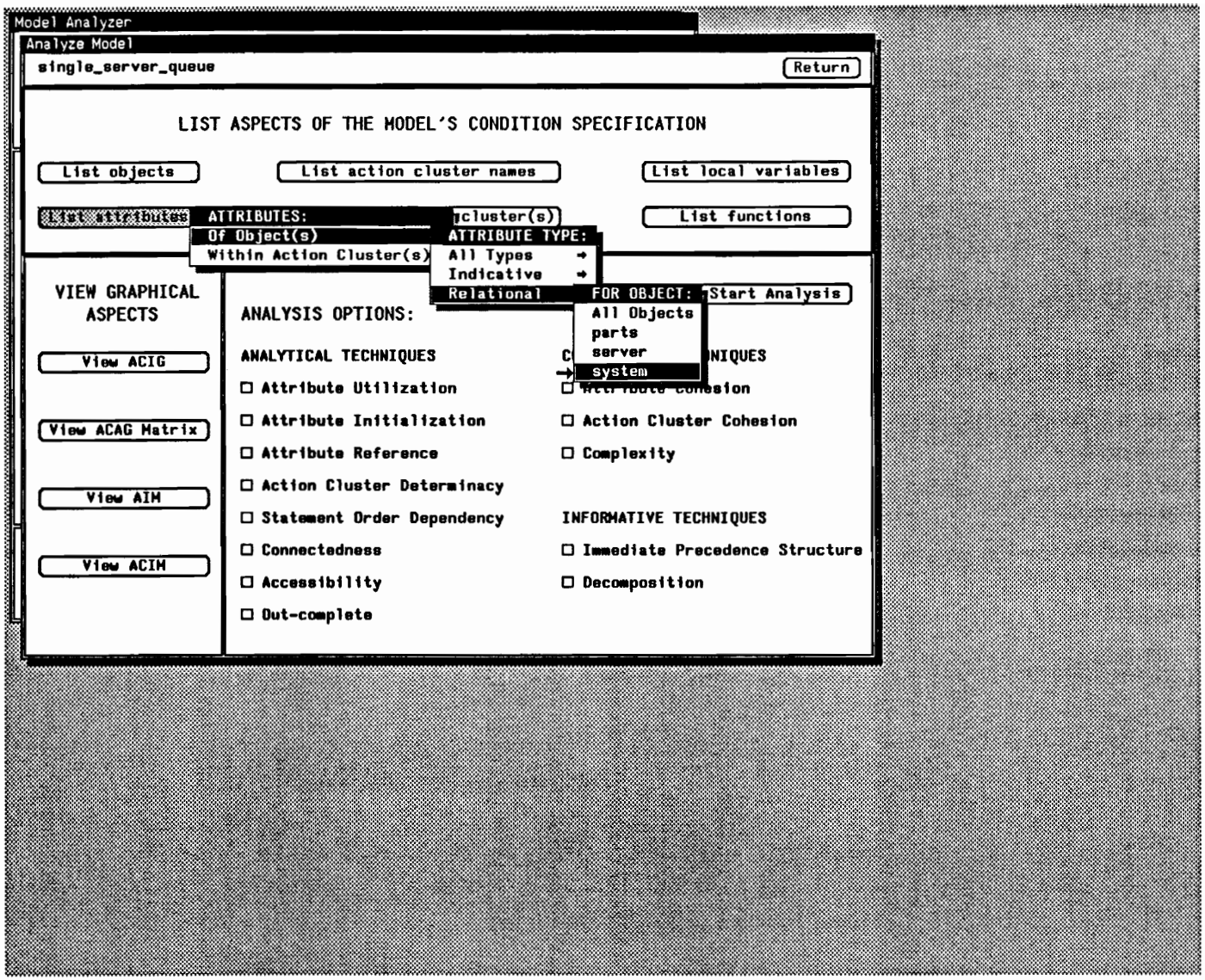


Figure 21. The "Attributes: Of Object(s)" Series of Pull-Right Menus

as choices. For example, if the modeler wants to display the relational attributes of the "system" object in the Single Server Queue Model, the modeler would follow the path highlighted in Figure 21 and select the "system" object.

In the "ATTRIBUTE TYPE" menu, the modeler chooses to view either the indicative or relational (CM type) attributes or all attributes of all types (CS and CM) for one or all objects in the model. Figure 22 shows the relational attributes of the "system" object in the Single Server Queue Model. Figure 23 illustrates the indicative attributes for all objects in the model. The attributes are separated by objects, and each attribute lists its specific indicative CM type. Choosing to see all types of attributes of all objects, shown in Figure 24, is the same as viewing the entire object specification of the model in an organized table. This table lists each attribute separated by its parent object and lists the CM and CS attribute type as well as the range of its index, if appropriate. If an attribute is enumerated, its legal enumerated values are listed below the object that contains the attribute.

The second purpose of the "List attributes" button, to determine the use of each attribute in the transition specification, is accomplished by following the second choice in the "ATTRIBUTES:" menu called "Within Action Cluster(s)". Following the right pointing arrow next to "Within Action Cluster(s)" displays a pull-right menu (Figure 25) entitled "ATTRIBUTE TYPE:" with four choices: (1) "All Types", (2) "Input", (3) "Output", and (4) "Control". Following any of the four choices displays the final menu called "FOR ACTION CLUSTER:", which lists "All Action Clusters" and the name of each action cluster in the model as choices.

The purpose of these menus is to list the type of use, or classification (input, output, or control), for each attribute in one or all action clusters in the model specification. For example, Figure 26 shows the control attributes of the "Termination"

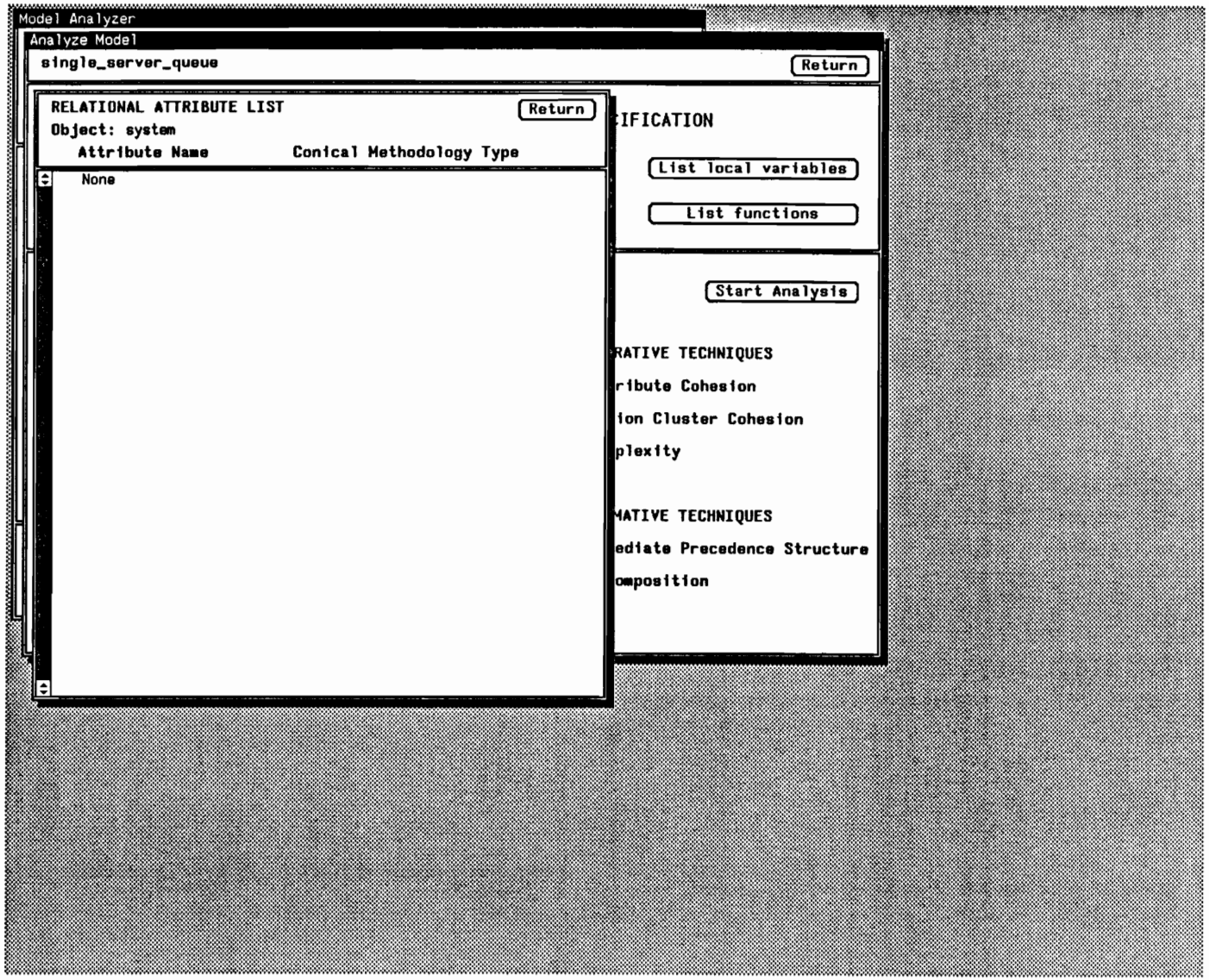


Figure 22. Relational Attributes of the "System" Object in the Single Server Queue

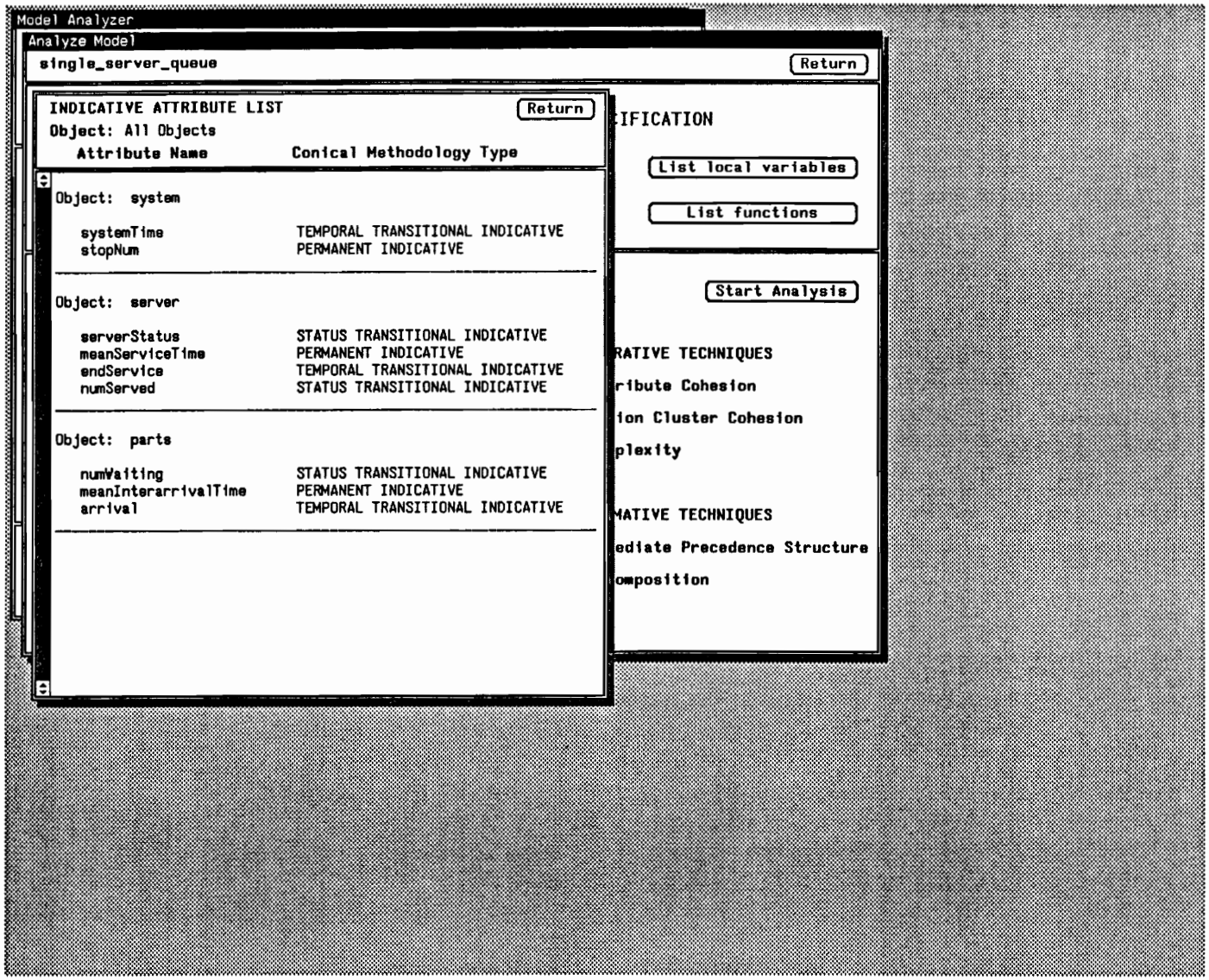


Figure 23. Indicative Attributes in All Objects of the Single Server Queue

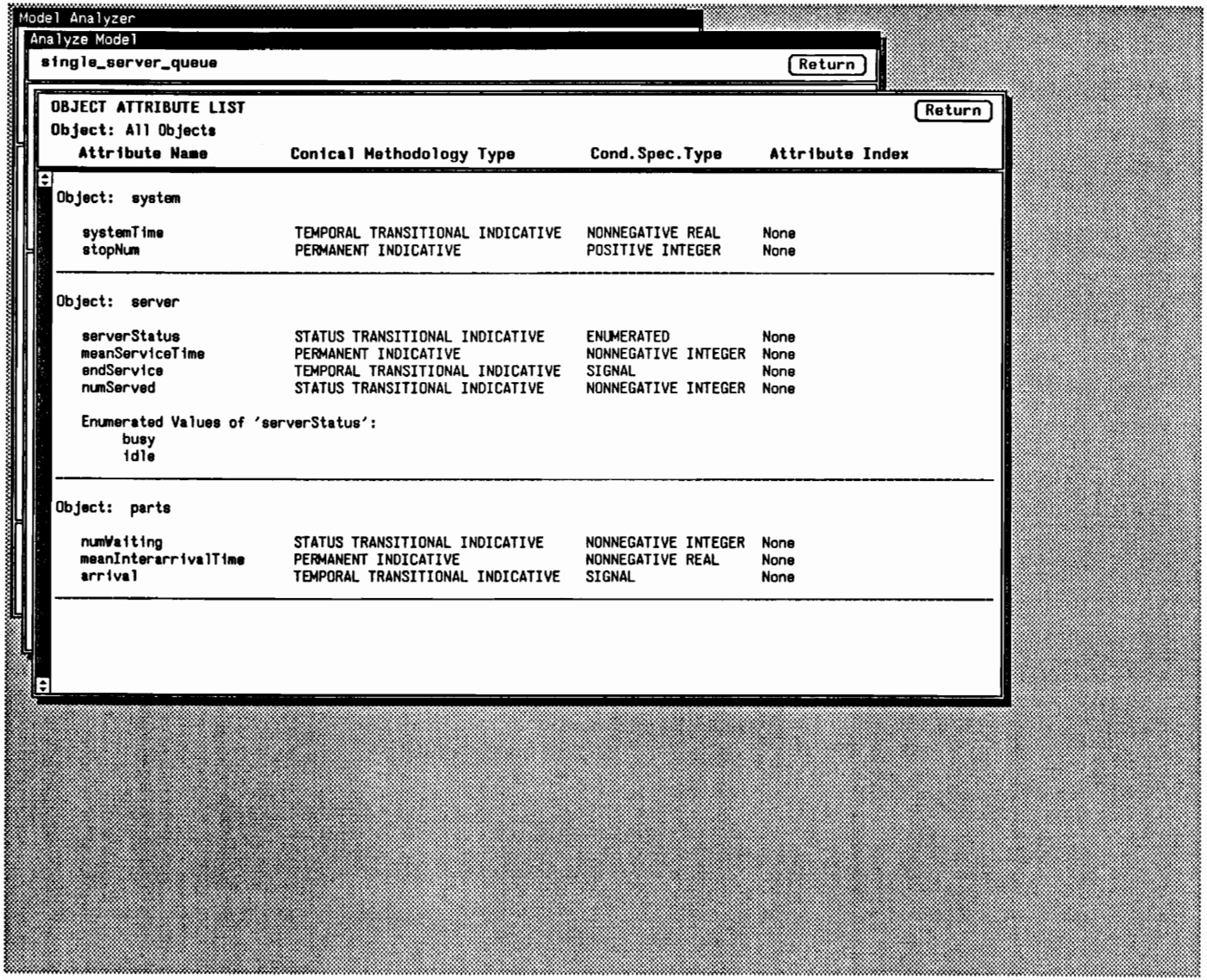


Figure 24. All Attributes of All Objects in the Single Server Queue

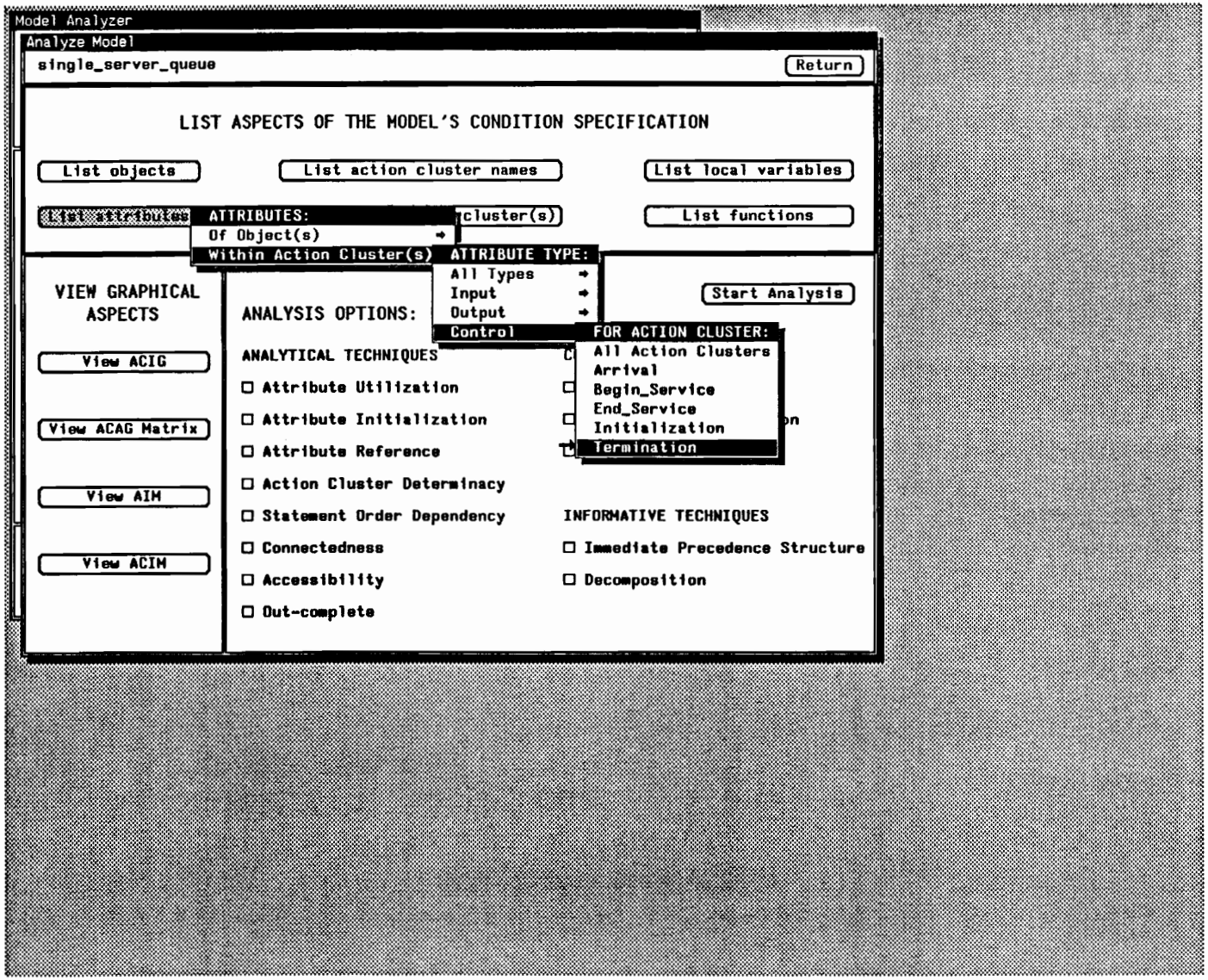


Figure 25. The "Attributes: Within Action Cluster(s)" Series of Pull-Right Menus

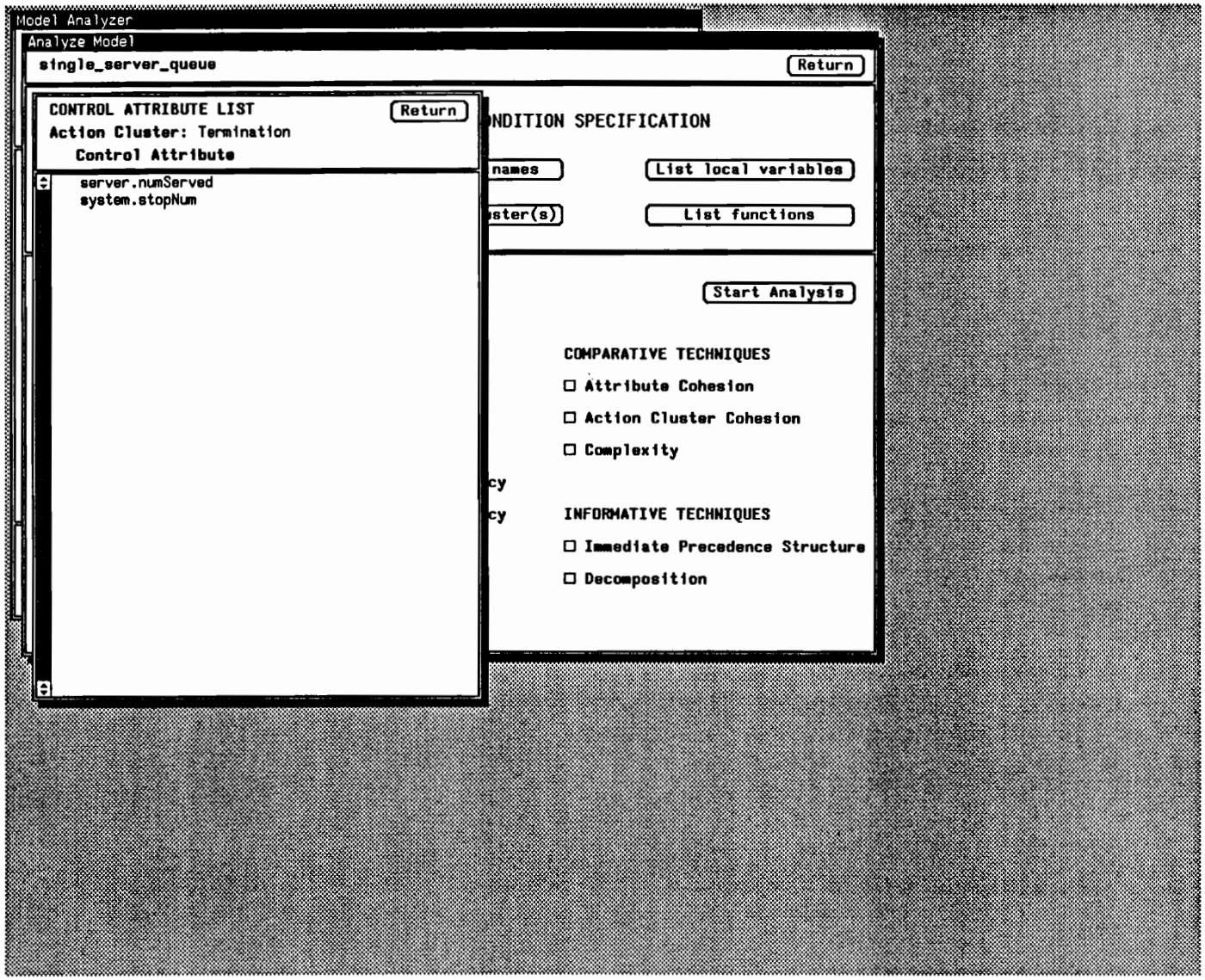


Figure 26. Control Attributes of "Termination" Action Cluster in Single Server Queue

action cluster in the Single Server Queue Model. Figure 27 lists the classification of all attributes within each action cluster in the model. Each AC is listed separately with its input, output, and control attributes following the AC name. The scrollbar must be used to view the entire list.

4.5.1.3 Action Clusters of the Model

The "List action cluster names" button gives the modeler an easy method of viewing the names of the action clusters in the model. When the button is selected, the Model Analyzer displays an alphabetical list of action cluster names, as illustrated in Figure 28.

4.5.1.4 Entire Action Clusters of the Model

To view the condition and actions of one or all action clusters in the CS of a model, the modeler must select the "List entire action cluster(s)" button. When the button is selected, a menu, shown in Figure 29, is displayed that allows the modeler to select "All Action Clusters" or the name of any action cluster in the model. Figure 30 shows the "Initialization" AC in the Single Server Queue Model. The condition is listed first followed by a series of actions. When the conditions and actions of all ACs are shown, each AC is separated with a horizontal line, displayed in Figure 31.

4.5.1.5 Local Variables in the Model Specification

The local variables used in the CS of a model is displayed by selecting the "List local variables" button. When the button is selected, a menu, shown in Figure 32, is displayed that lists "All Action Clusters" and the names of each AC in the model as choices. The modeler may want to display the local variables for a particular AC or all

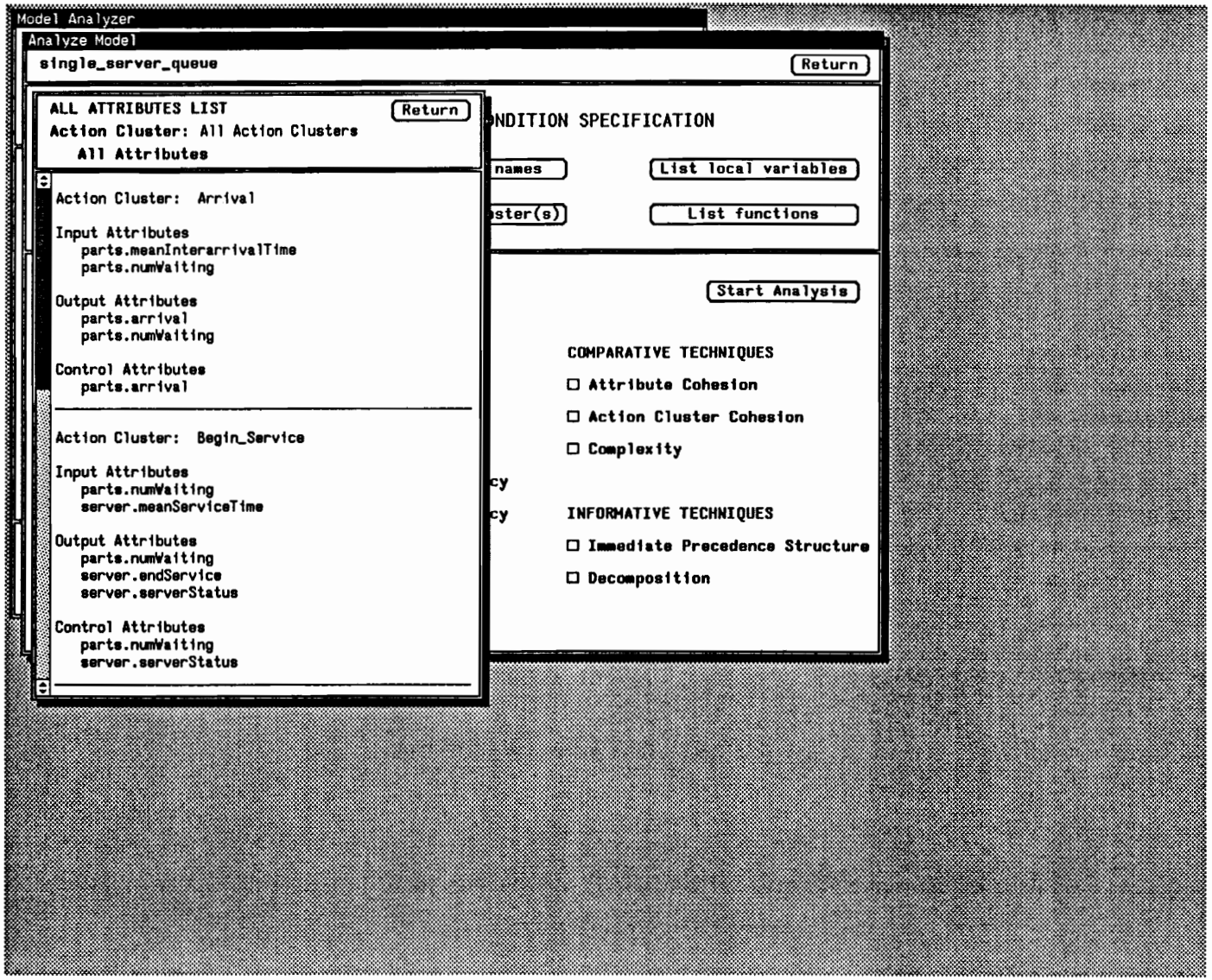


Figure 27. Classification of All Attributes of All Action Clusters in Single Server Queue

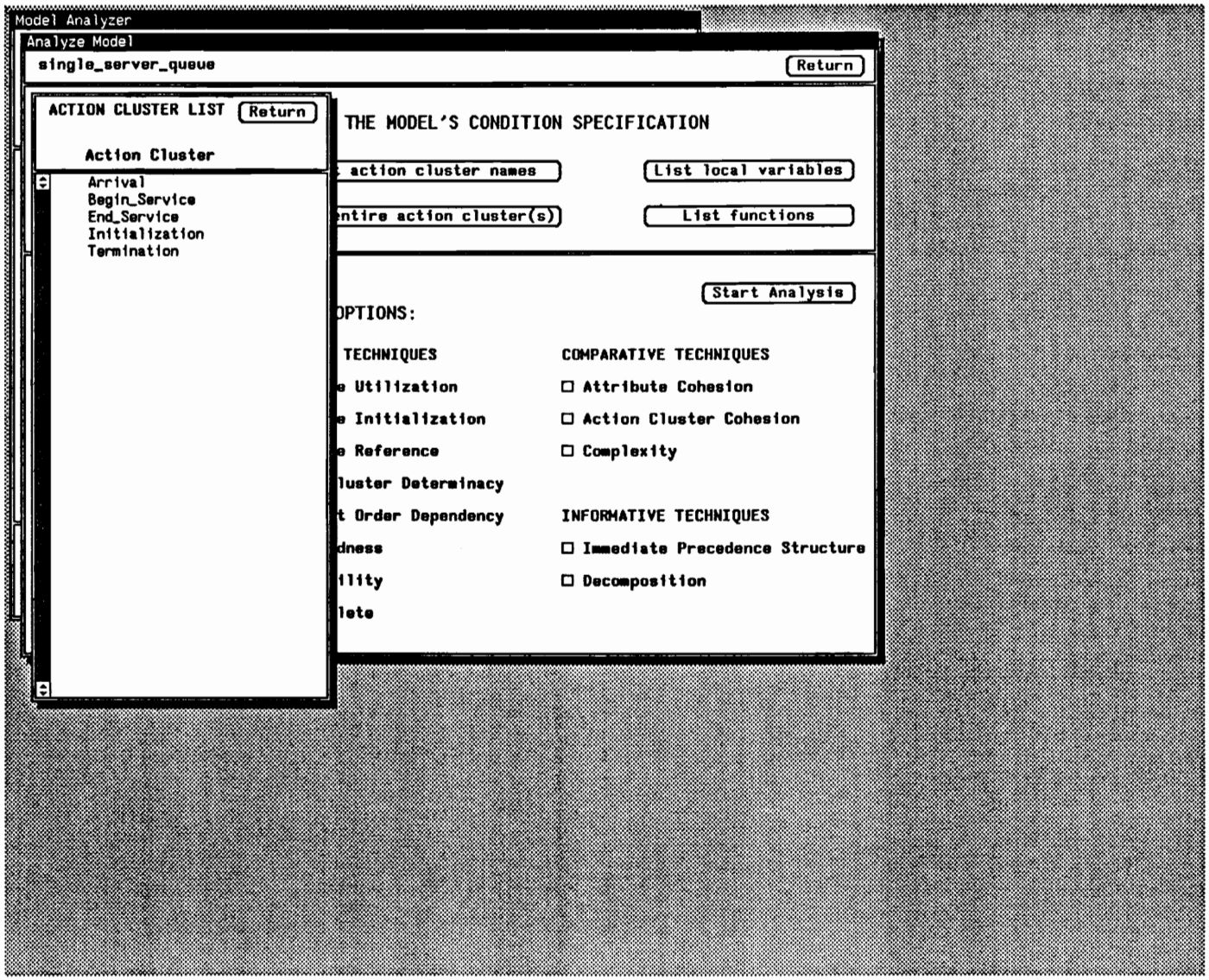


Figure 28. List of All Action Cluster Names in the Single Server Queue

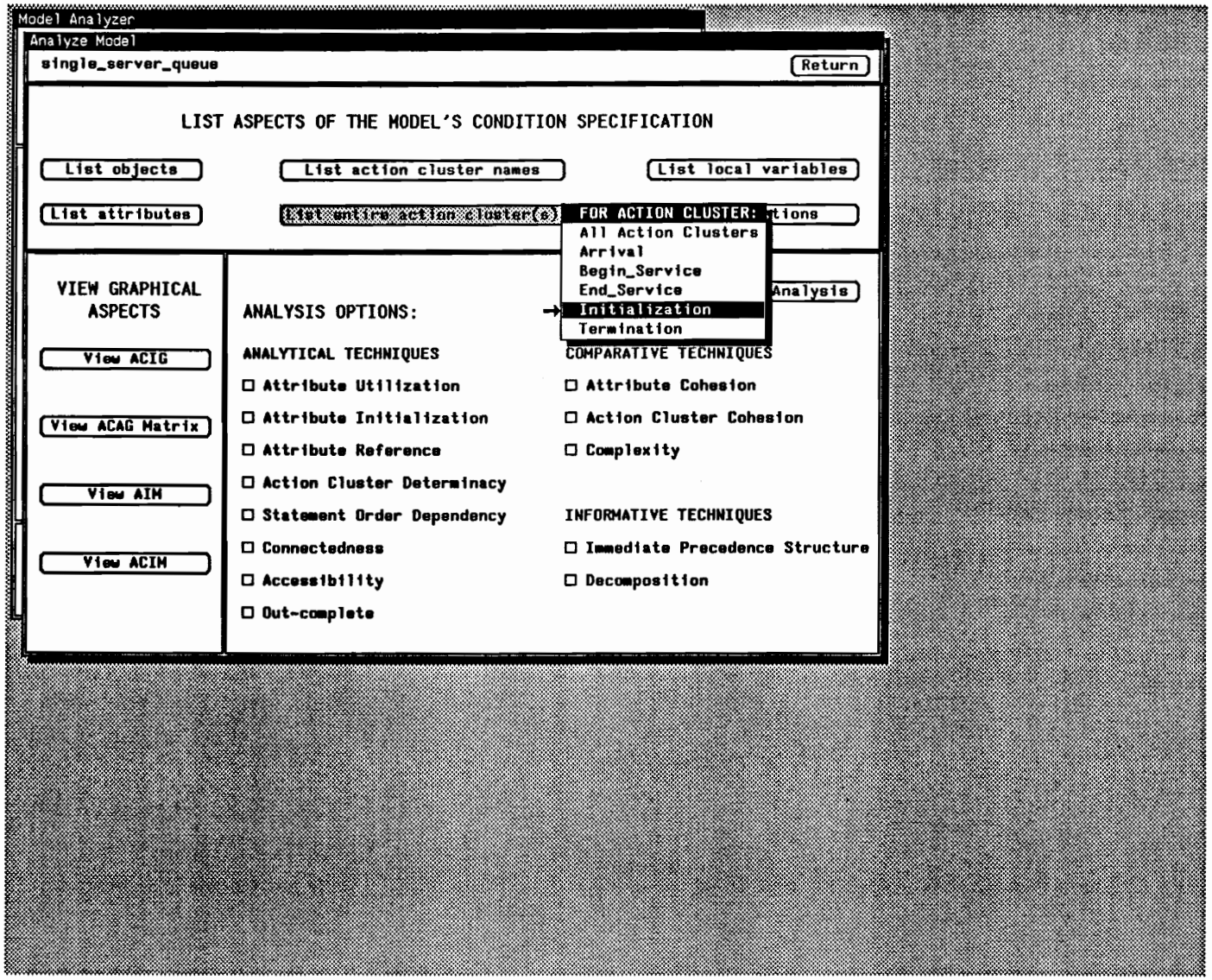


Figure 29. The Menu of the "List entire action cluster(s)" Button

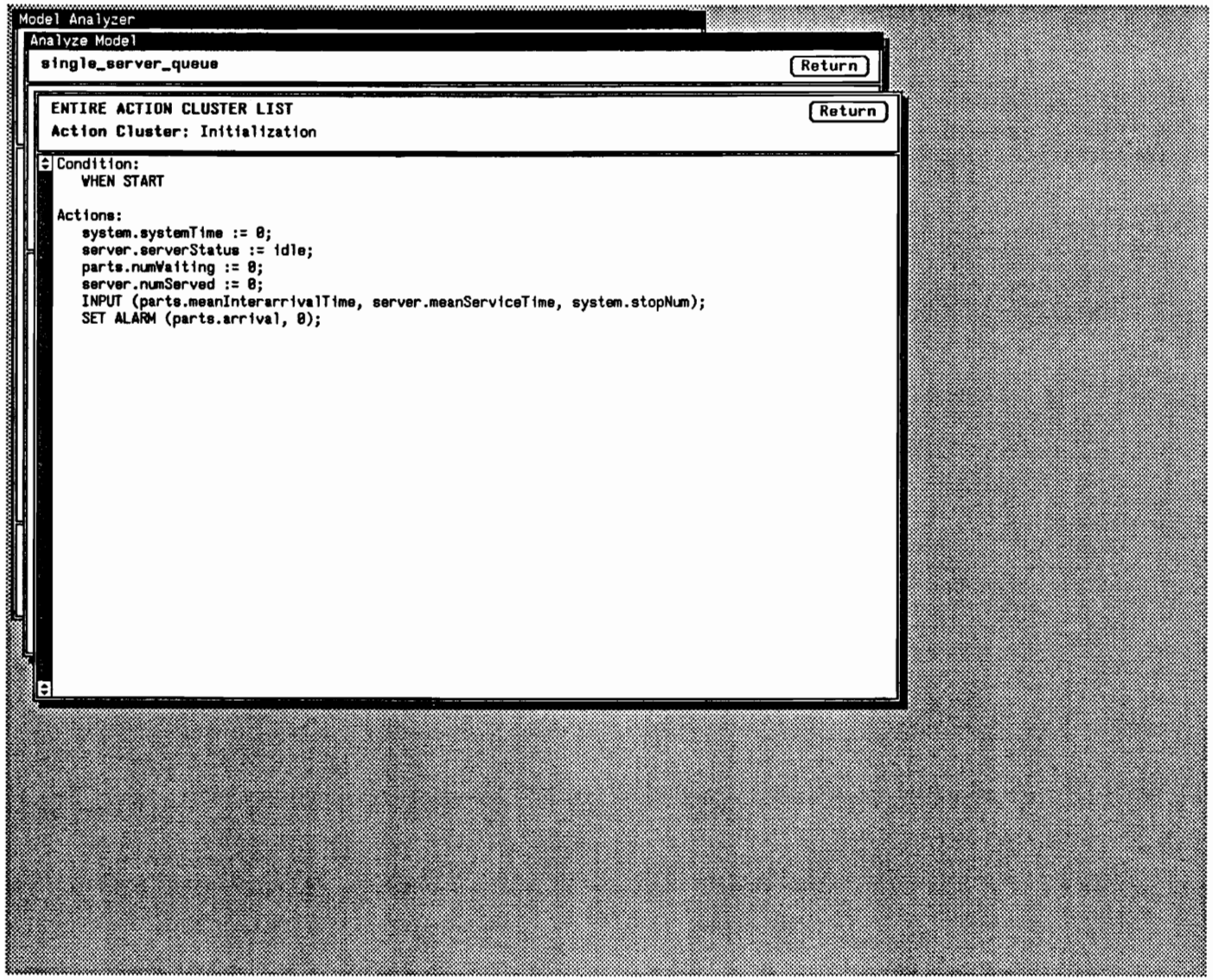


Figure 30. The Entire "Initialization" Action Cluster in the Single Server Queue

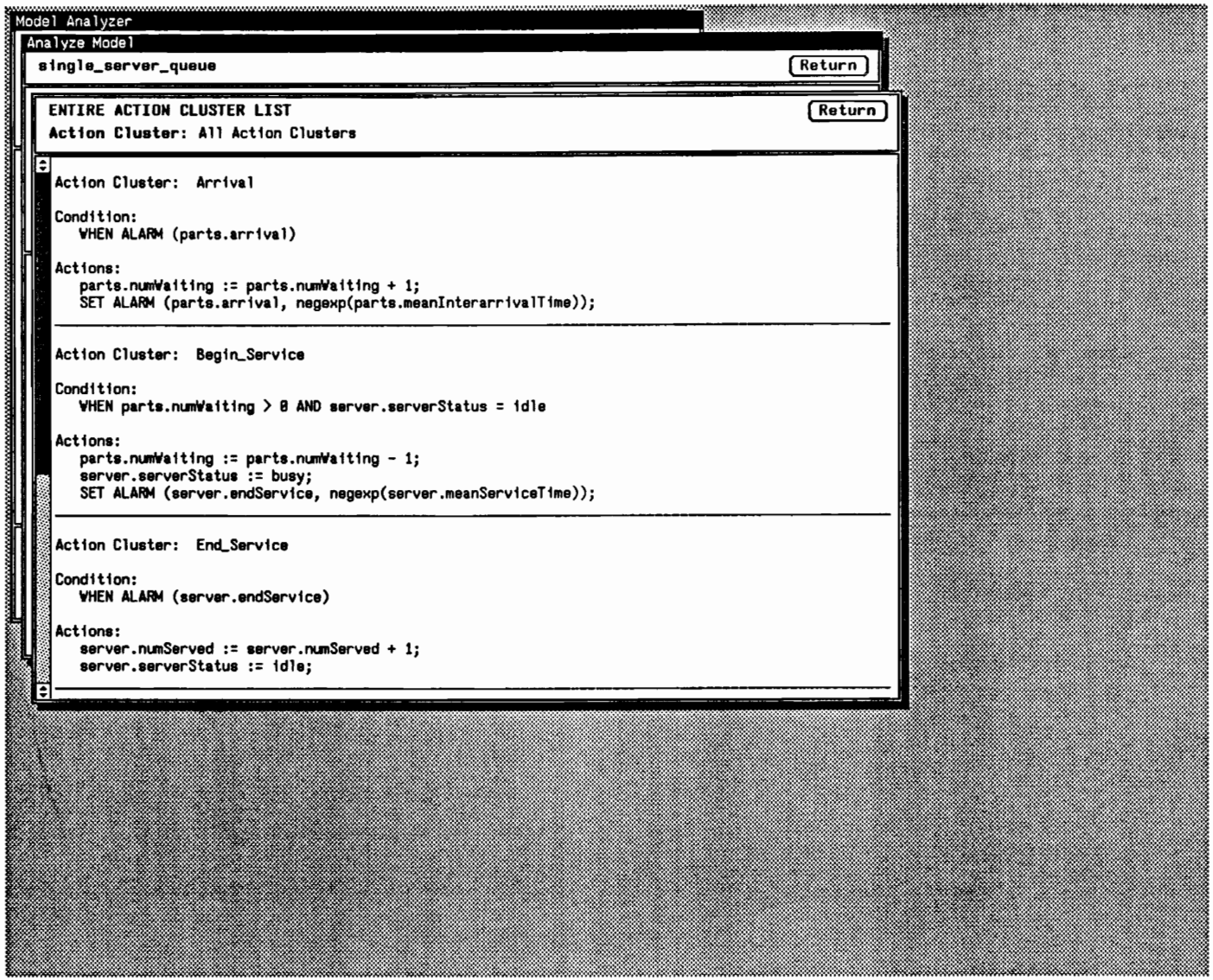


Figure 31. The Conditions and Actions of All Action Clusters in Single Server Queue

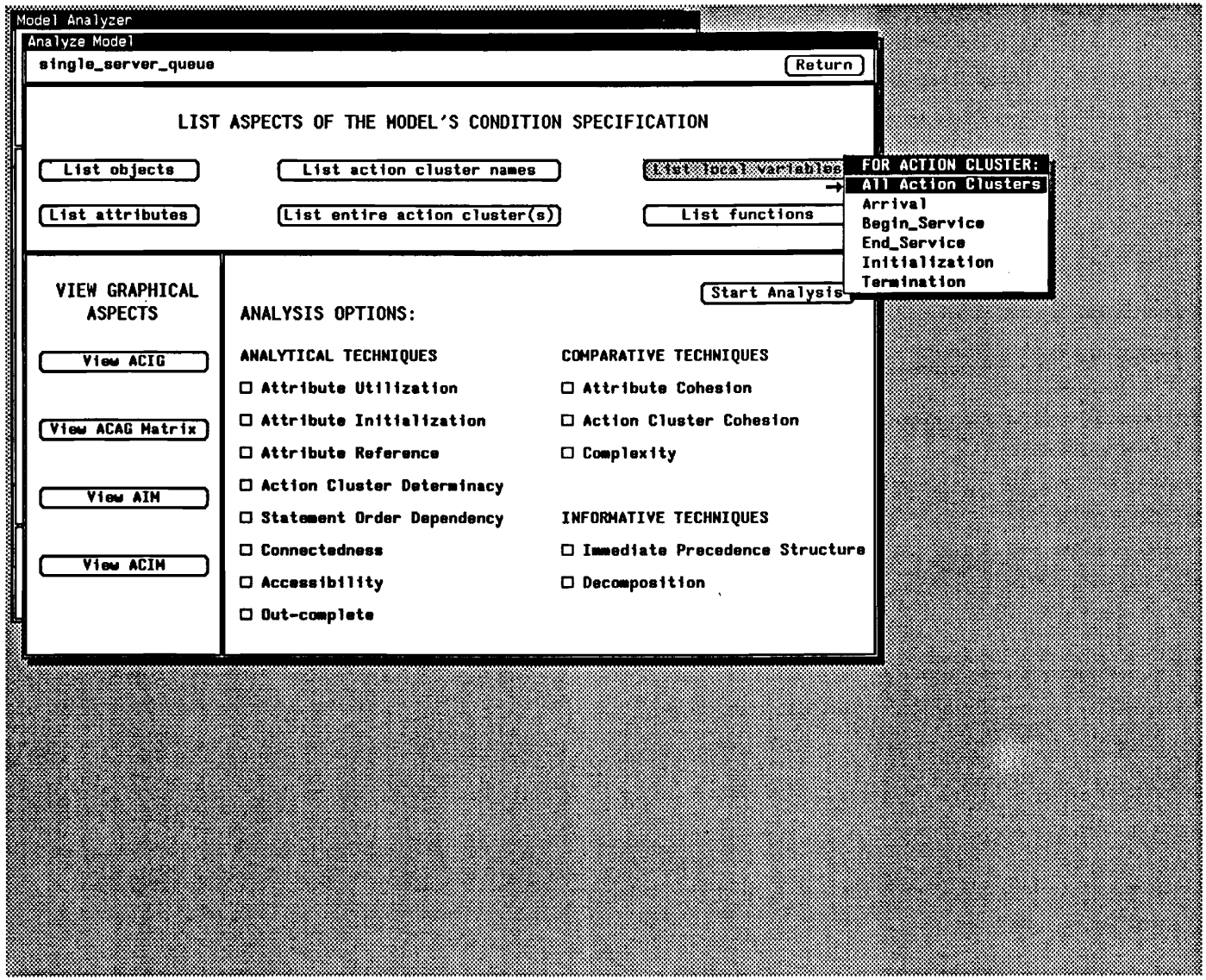


Figure 32. The Menu of the "List local variables" Button

ACs. Figure 33 shows the Model Analyzer when the modeler views all the local variables used in the model. Each AC lists the variables that are local to its own AC. The analyzer defines a local variable as any identifier that is not defined as an object, attribute, or enumerated value in the object specification of the CS.

4.5.1.6 Functions in the Model Specification

To display the functions used in the CS of a model, the modeler must use the "List functions" button. When the button is selected, a menu, shown in Figure 34, appears that is identical with the menu for the "List entire action cluster(s)" and "List local variables" buttons. A function is defined as any non-keyword followed by a list of zero or more parameters enclosed in a set of parentheses. The functions used in a particular AC or all ACs may be displayed. Figure 35 shows the name of the functions used in each AC of the Single Server Queue Model.

4.5.2 Graphical Aspects of Model Analyzer

To view the specification of the model graphically, the four buttons in the lower left corner of the Analyze Model panel, shown in Figure 18, are used. Graphical representations enhance the visualization of the model and its interactions among model components. This section of the Model Analyzer follows the ancient saying: "a picture is worth a thousand words." The graphical representations produced in the analyzer consist of digraphs and matrices from four components of the model: (1) action cluster incidence graph (ACIG), (2) action cluster attribute graph (ACAG), (3) attribute interaction matrix (AIM), and (4) action cluster interaction matrix (ACIM). These four graphical features are described in the following sections.

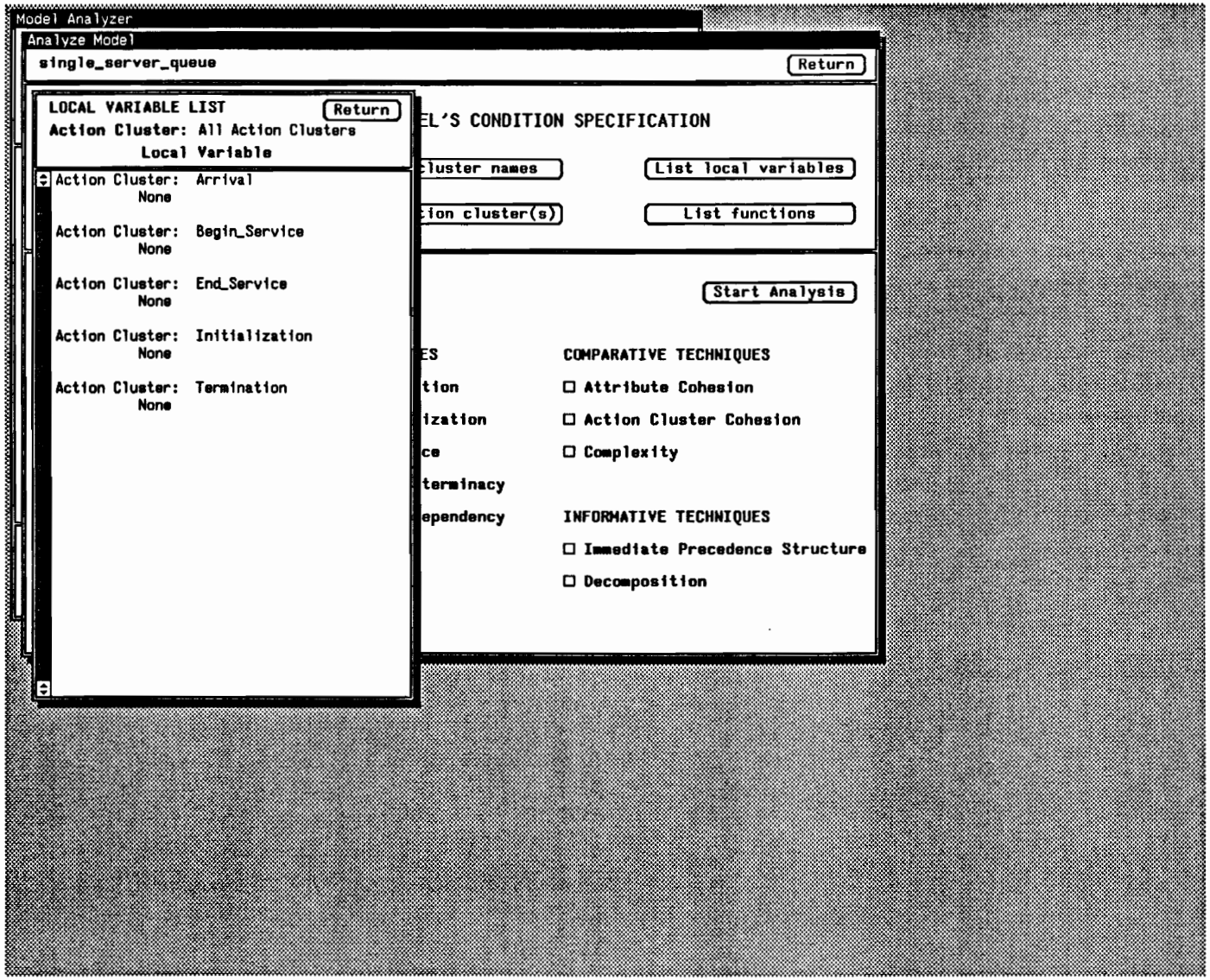


Figure 33. List of Local Variables in All Action Clusters of Single Server Queue

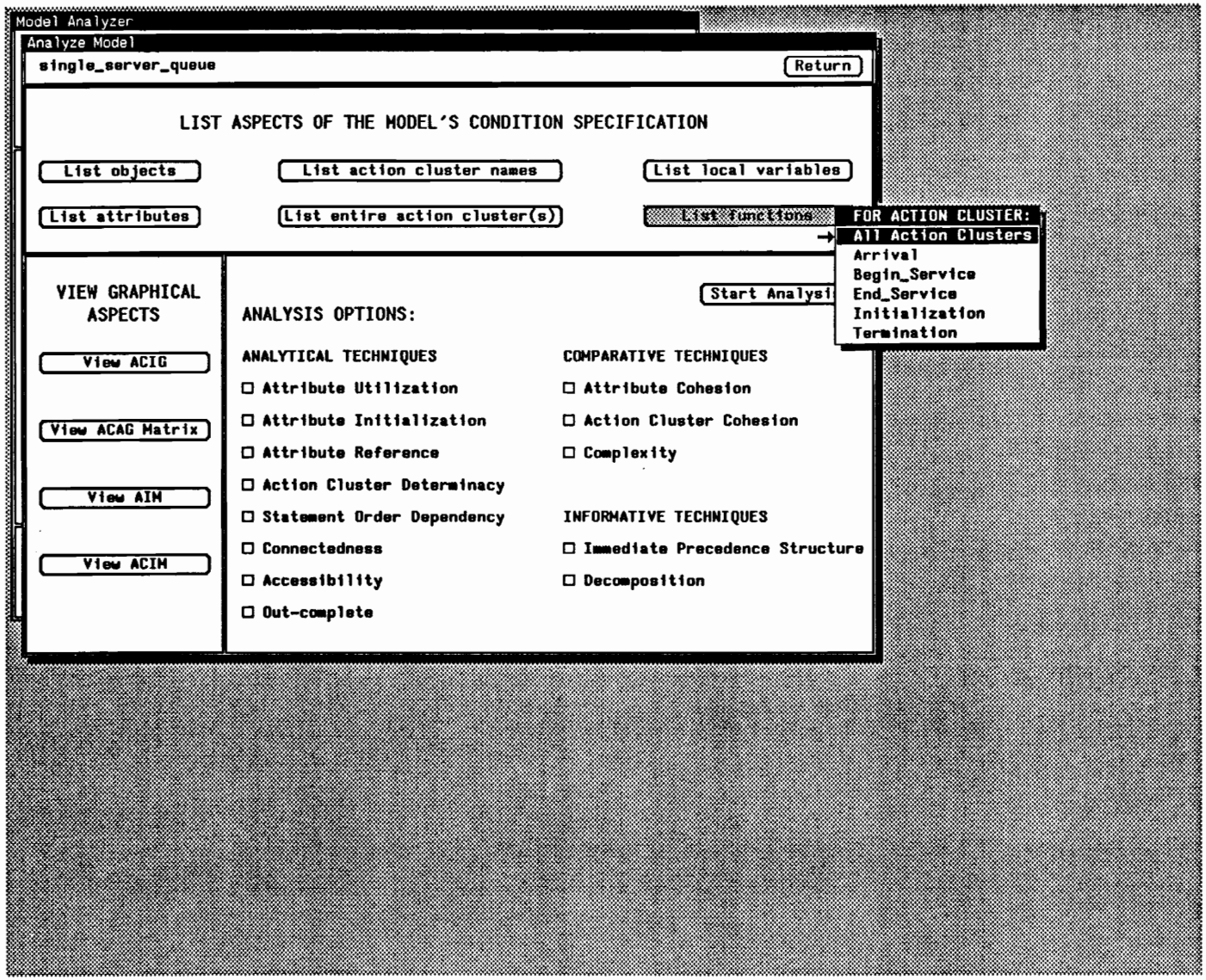


Figure 34. The Menu of the "List functions" Button

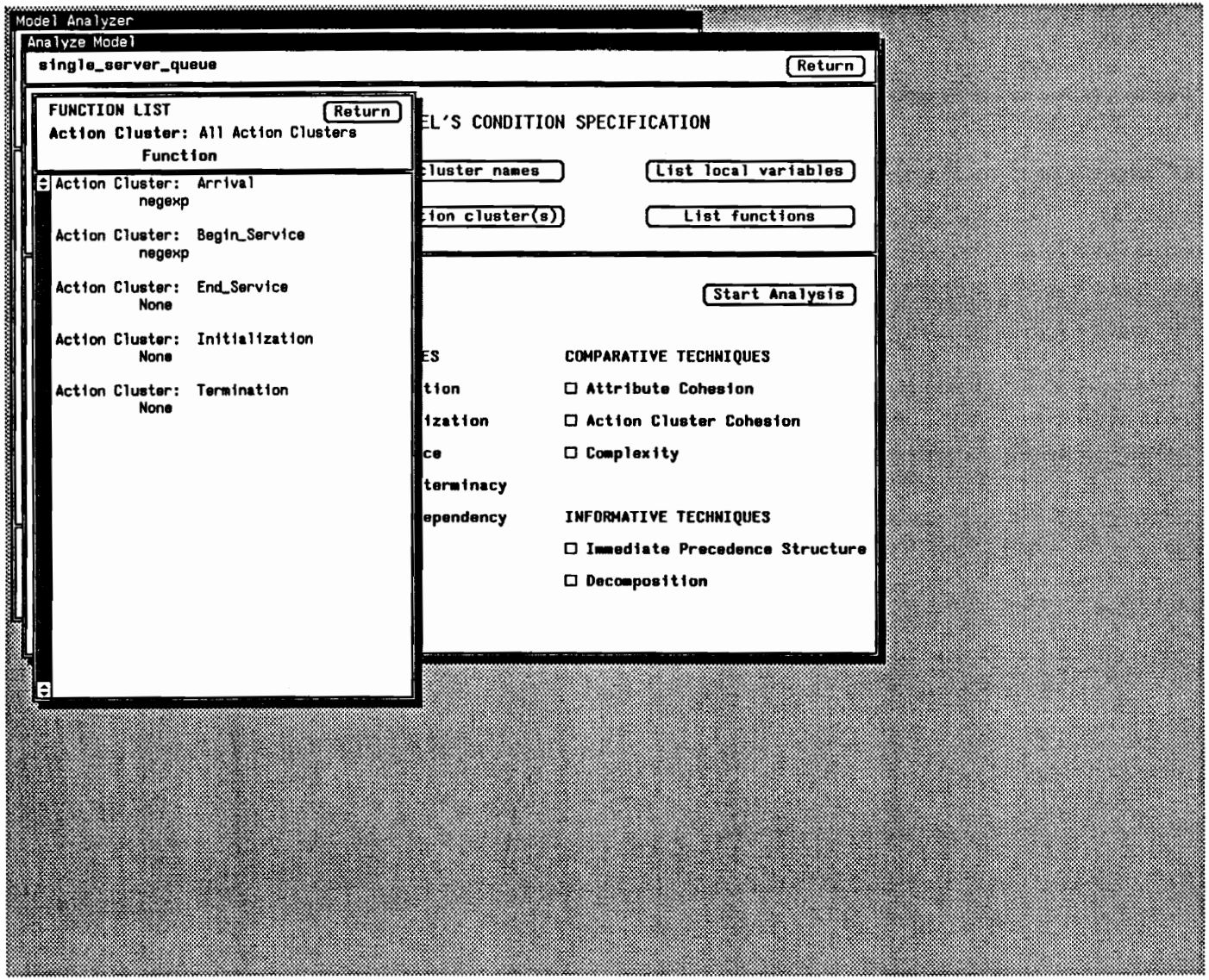


Figure 35. List of Functions in All Action Clusters of Single Server Queue

4.5.2.1 Action Cluster Incidence Graph

The action cluster incidence graph, as described in Section 3.1.2, depicts the interactions among action clusters in the specification of the model. The top button, entitled "View ACIG", is used to view the ACIG in various graphical forms. When this button is selected, a pull-right menu, shown in Figure 36, called "ACIG TYPE:" appears with two choices: (1) "Simplified" and (2) "Unsimplified". Following the right pointing arrow for either choice displays another menu entitled "ACIG FORM:", which has three choices that allow the modeler to display the ACIG as a matrix, circular graph, or linear graph. The unsimplified ACIG contains all the edges that are produced from the algorithm in Section 3.1.2. The simplified ACIG contains a subset of the edges in the unsimplified ACIG. Any edges that depict interactions among two action clusters that may never occur are removed in the simplified ACIG. Chapter 5 describes a method of determining these infeasible edges. The graphical forms of the unsimplified ACIG are discussed in Section 4.5.2.1.1, and Section 4.5.2.1.2 describes the graphical forms of the simplified ACIG.

4.5.2.1.1 Unsimplified Action Cluster Incidence Graph

In the Model Analyzer, the unsimplified ACIG can be viewed as three different graphical representations: (1) matrix, (2) circular graph, and (3) linear graph. Figure 37 shows the unsimplified ACIG as a matrix for the Single Server Queue Model. The total number of edges in the ACIG and the legend are listed in the top section of the panel. Each row in the matrix is labeled with a name of an action cluster preceded by a corresponding number. These corresponding numbers are also listed along the top of each column in the matrix. The matrix is filled with either a *T*, which represents a time-based edge, or an *S*, which represents a state-based edge. Looking at the matrix in Figure

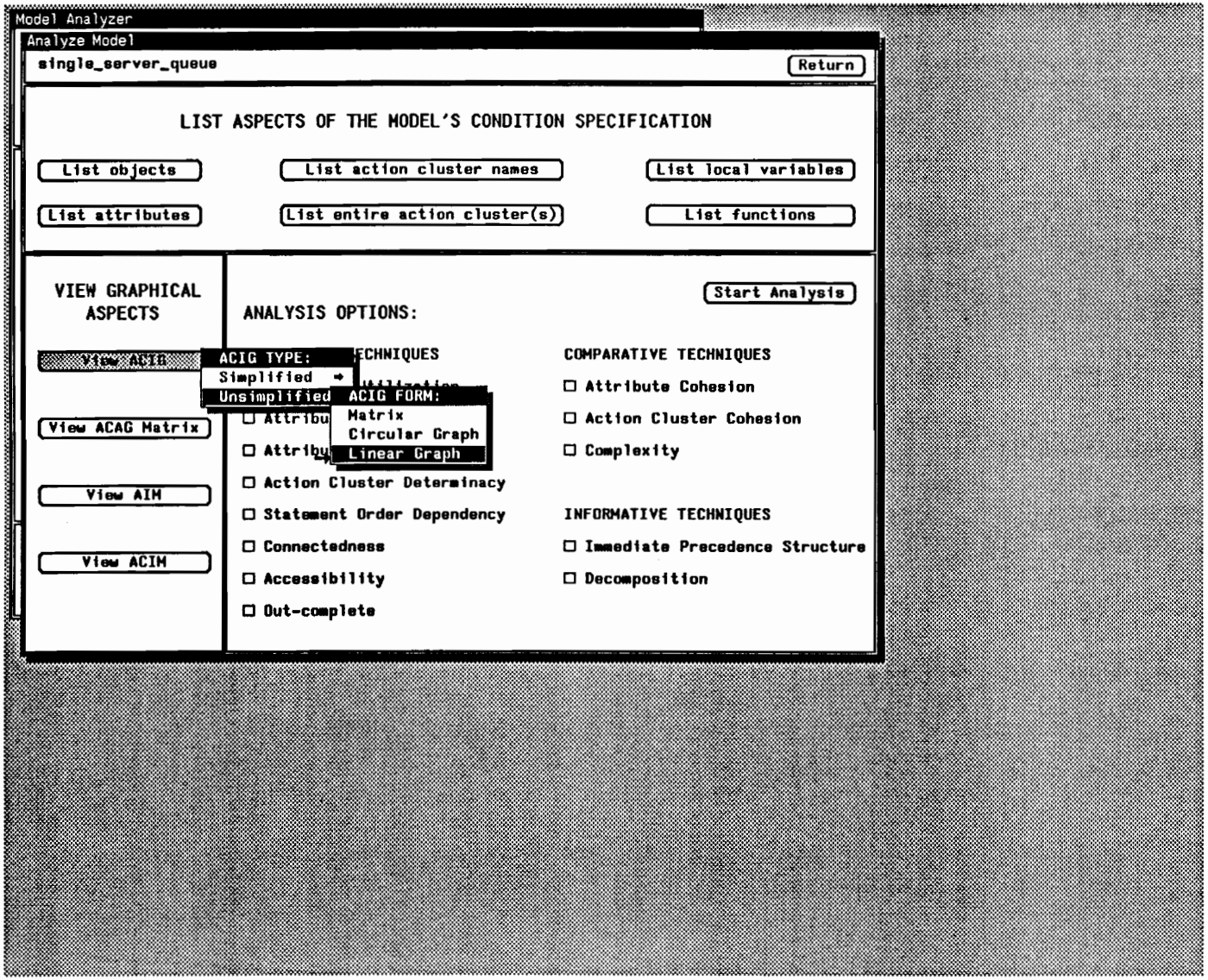


Figure 36. The Series of Menus of the "View ACIG" Button

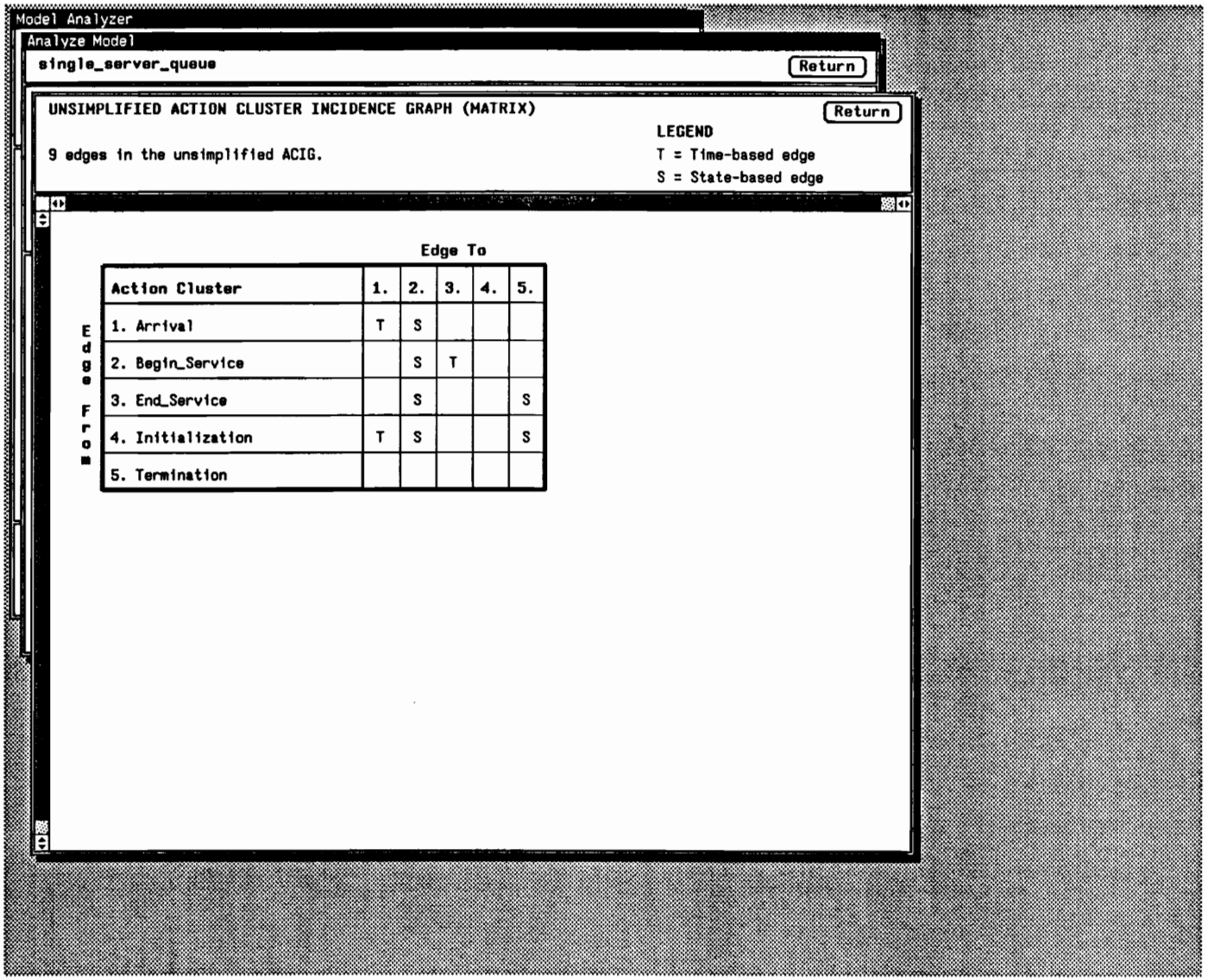


Figure 37. Matrix Form of the Unsimplified Action Cluster Incidence Graph

37, the *T* in the second row and third column symbolizes a time-based edge from the "Begin_Service" AC to the "End_Service" AC; whereas, the *S* in the first row and second column symbolizes a state-based edge from the "Arrival" AC to the "Begin_Service" AC.

The Model Analyzer also allows the modeler to view the unsimplified ACIG as a circular graph, as illustrated in Figure 38. Like the matrix, the total number of edges and the legend are listed in the top section of the panel. Figure 38 is called a circular graph because the action clusters, represented as labeled boxes, are arranged in a circular pattern, which allows the entire graph, no matter how many ACs it may have, to be shown on one screen. Each edge joining two ACs is shown as a solid line, representing a state-based edge, or as a dashed line, representing a time-based edge. The circular graph does a good job of representing models that have twelve or fewer ACs. If a model has more than twelve ACs, the edges in the circular graph between two AC boxes might intersect other AC boxes.

The last method of graphically representing the unsimplified ACIG is the traditional linear graph seen in [Nance and Overstreet 1986, 1987a, b]. The unsimplified ACIG for the Single Server Queue Model is shown in Figure 39 as a linear graph. Like the other two representations shown above, the total number of edges and the legend are listed in the top section of the panel. The action clusters in the model are represented as labeled boxes that are arranged in a vertical linear fashion. The interactions among the ACs are drawn as solid or dashed lines, like the circular graph. A linear graph that represents a model with greater than nine ACs does not fit entirely on the screen of the Model Analyzer, and the scrollbar must be used to see the bottom portions of the graph. The ACs in the linear graph are arranged to represent a top-to-bottom flow of edges. The "Initialization" AC is always listed as the top AC in the graph. The next AC has an

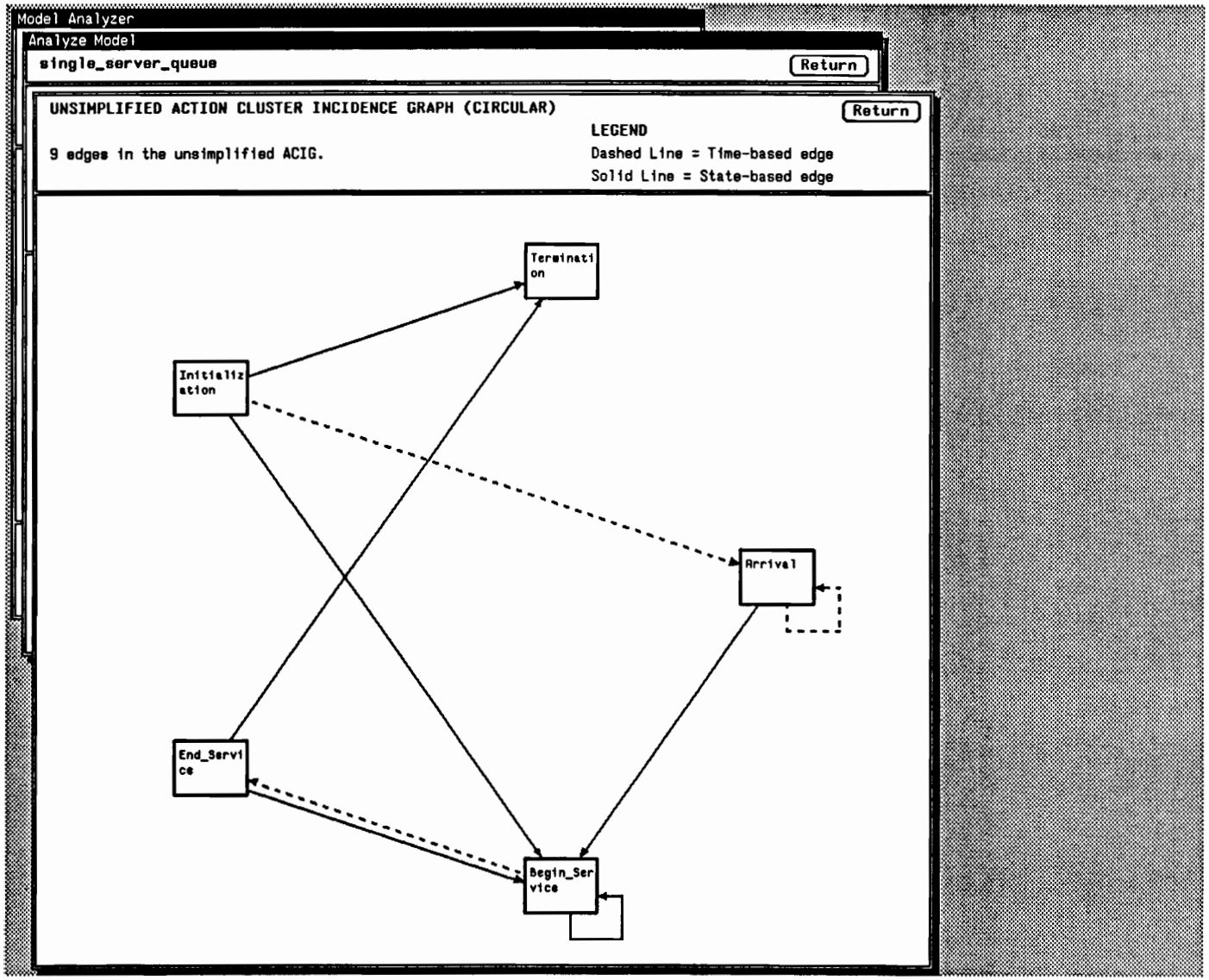


Figure 38. Circular Form of the Unsimplified Action Cluster Incidence Graph

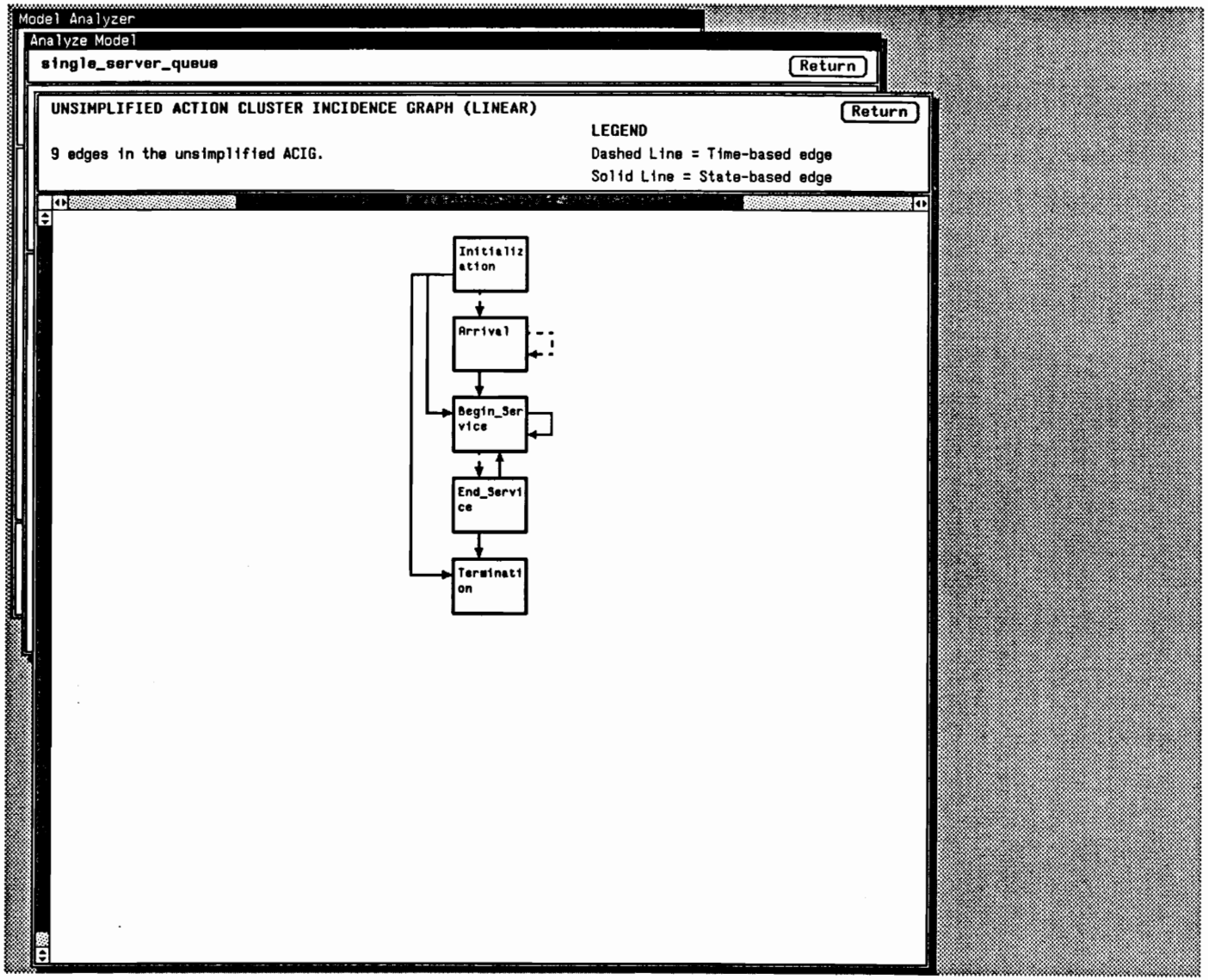


Figure 39. Linear Form of the Unsimplified Action Cluster Incidence Graph

arriving edge that comes from the "Initialization" AC. The ACs are arranged so most ACs will each have an edge pointing to the AC below it, thus producing the top to bottom flow of edges.

4.5.2.1.2 Simplified Action Cluster Incidence Graph

The simplified action cluster incidence graph for a model contains the same action clusters as the unsimplified ACIG, but only a subset of its edges. The Model Analyzer can graphically view the simplified ACIG in the same three representations as the unsimplified ACIG: (1) matrix, (2) circular graph, and (3) linear graph.

When the simplified ACIG is viewed for the first time by the Model Analyzer, the picture of the graph or matrix is exactly the same as the unsimplified ACIG. The differences between the two occur in the top section of each panel. Figure 40 shows the simplified Single Server Queue ACIG as a linear graph that is viewed for the first time in the Model Analyzer. The top section displays the number of edges in the simplified ACIG, the number of edges removed from the unsimplified ACIG, the percent reduction of edges, and the legend. The top section in the simplified ACIG also contains two buttons not present in the unsimplified ACIG panel: (1) "Redraw Graph" and (2) "Remove edge(s)".

To create the simplified ACIG from the unsimplified ACIG, the modeler must remove the infeasible edges from the graph. This process is done manually in the Model Analyzer by selecting the "Remove Edge(s)" button. When the button is selected, a panel appears to the right of the graph that lists all the edges in the graph, as displayed in Figure 41. The modeler must select the edges in the list, one at a time, that should be removed from the graph. As each edge is selected, that edge disappears in the list but remains in the graph. Figure 42 shows the list of edges in the ACIG after three edges

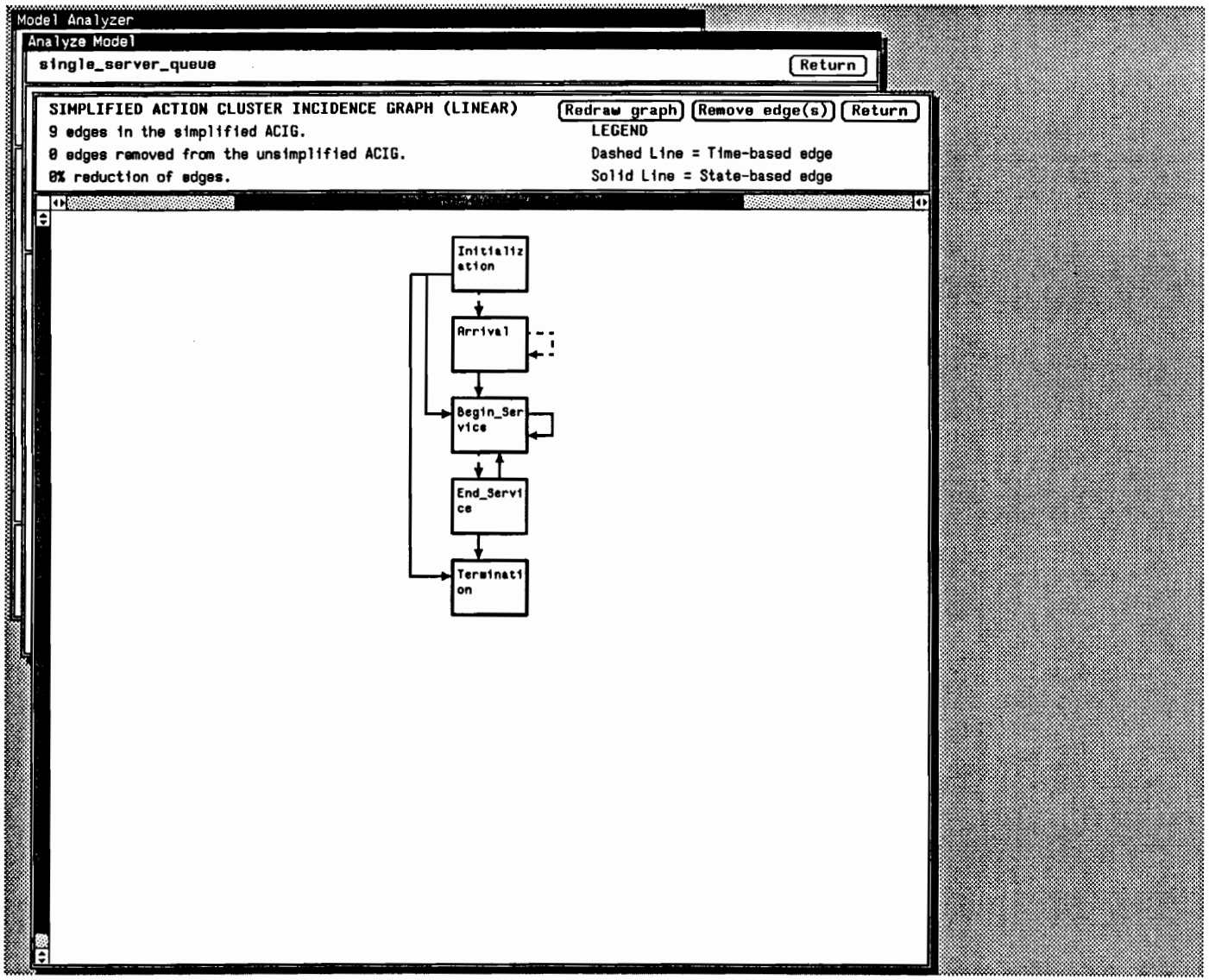


Figure 40. Single Server Queue Simplified Action Cluster Incidence Graph Before Infeasible Edges Are Removed

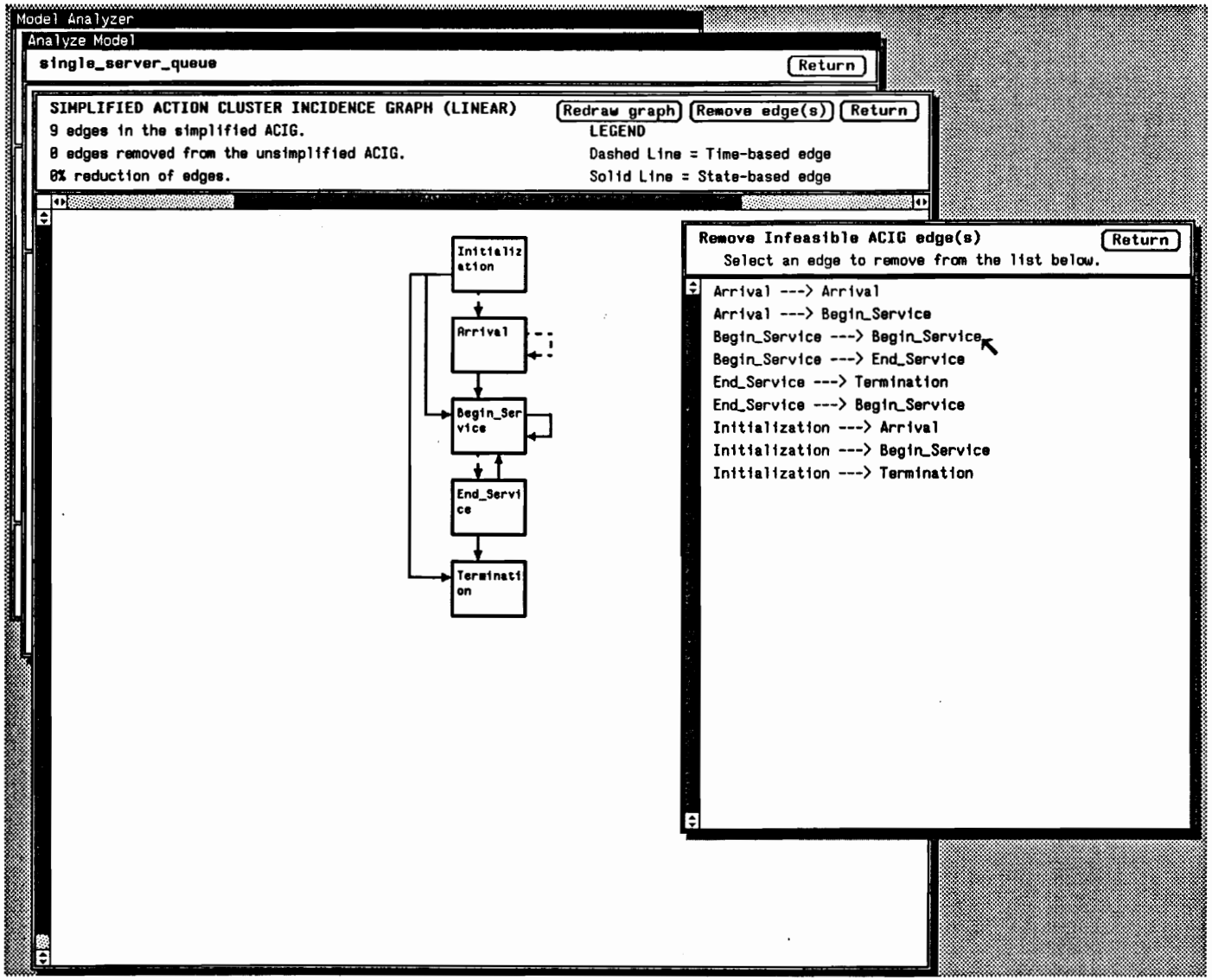


Figure 41. List of Edges in Single Server Queue Simplified Action Cluster Incidence Graph Before Any Edges Are Removed

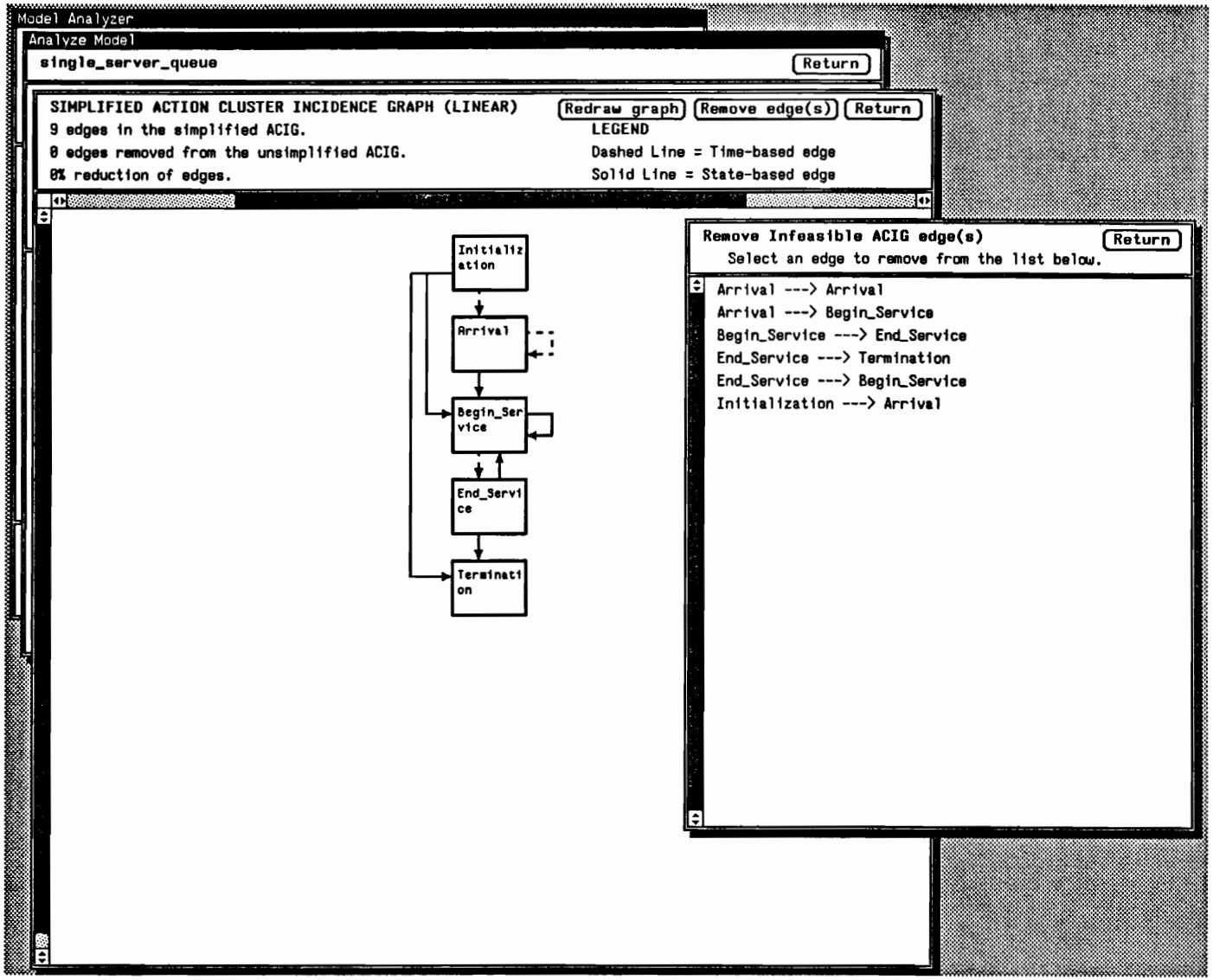


Figure 42. List of Edges in Single Server Queue Simplified Action Cluster Incidence Graph After Three Edges Are Removed

were removed from the list in Figure 41, but notice that the graph remains the same. Only after selecting the "Redraw graph" button are the edges finally removed from the graph, as shown in Figure 43. Also at this time, the statistics on the number of edges in the simplified ACIG, the number of edges removed from the unsimplified ACIG, and the percent reduction of edges are updated.

When the edges are removed from the ACIG, this reduction is remembered and recorded in the simplified ACIG relation of the database. The next time the simplified ACIG is viewed in any of the three graphical formats, the graph or matrix does not contain the deleted edges.

The edges of the ACIG also may be deleted from the simplified ACIG when viewed as a matrix or circular graph. Figure 44 shows the simplified ACIG as a matrix, and Figure 45 shows the simplified ACIG as a circular graph. The matrix and circular graph depicts the same edges and statistics as the linear graph in Figure 43.

4.5.2.2 Action Cluster Attribute Graph

The action cluster attribute graph, which depicts the interactions between action clusters and attributes, is viewed as a matrix by selecting the "View ACAG Matrix" button in the Model Analyzer. Since the ACAGs of most models consist of many edges and nodes, the Model Analyzer views the ACAG as a matrix, which organizes the information into a structured table, instead of viewing a cluttered graph. Figure 46 illustrates the ACAG matrix for the Single Server Queue Model. The rows in the matrix list the model attributes, and the columns are labeled with action clusters. The positions in the matrix are filled with combinations of four symbols: (1) *I*, (2) *C*, (3) *O*, and (4) *T*. Each symbol represents how the attribute in that row is used in the action cluster of that column. For example, Figure 46 shows a *C* and a *T* in the first row and column of the

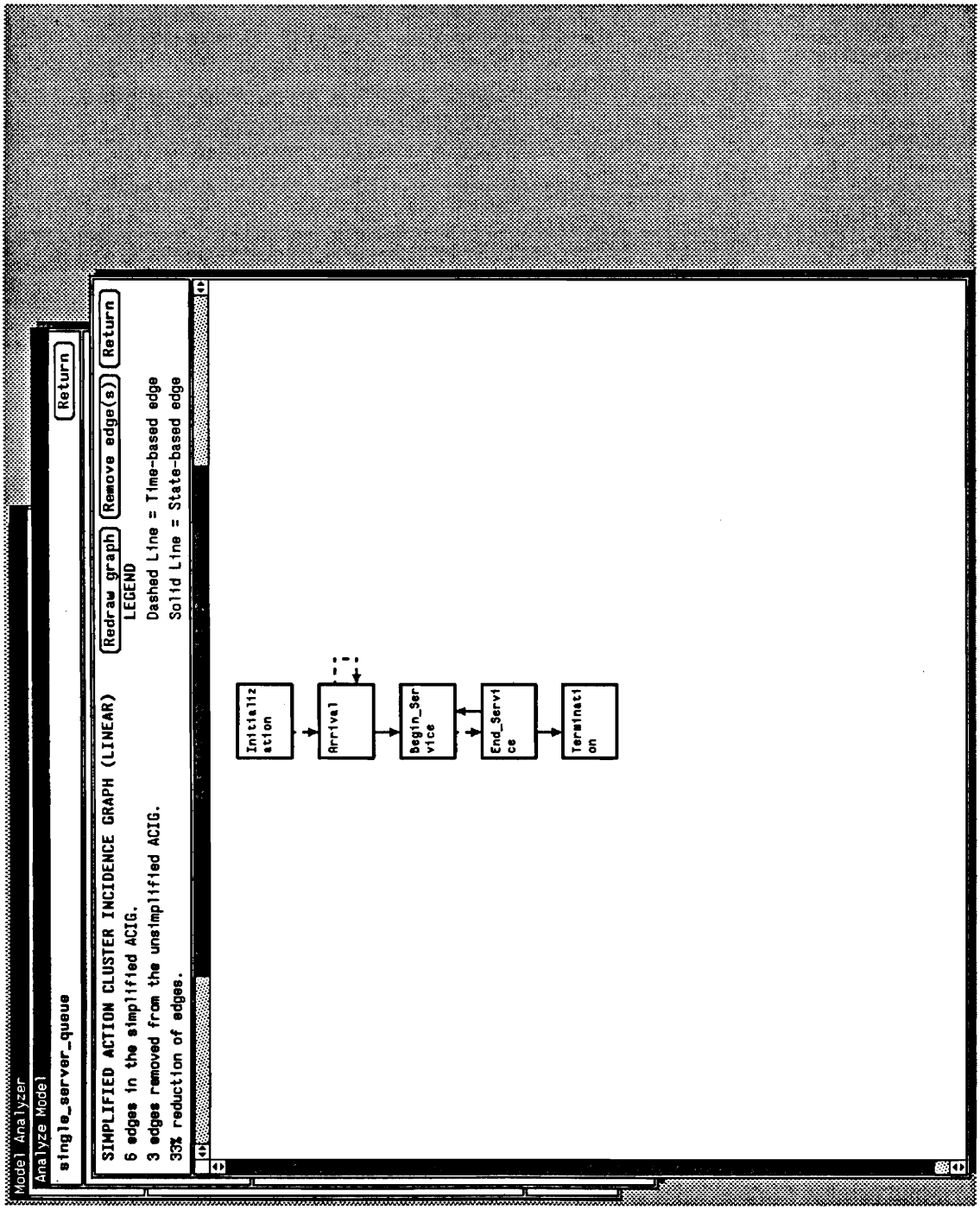


Figure 43. Linear Form of Simplified Action Cluster Incidence Graph

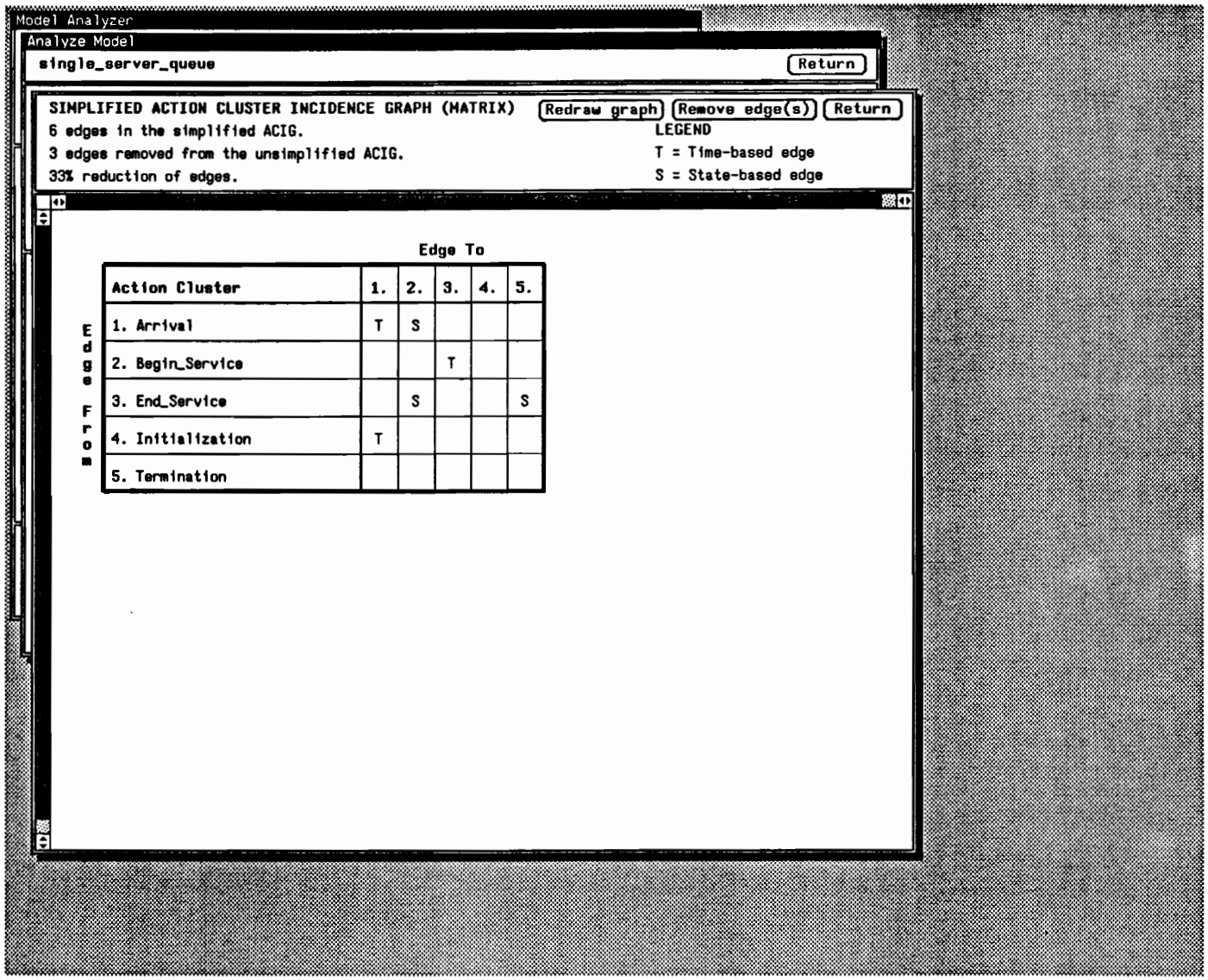


Figure 44. Matrix Form of Simplified Action Cluster Incidence Graph

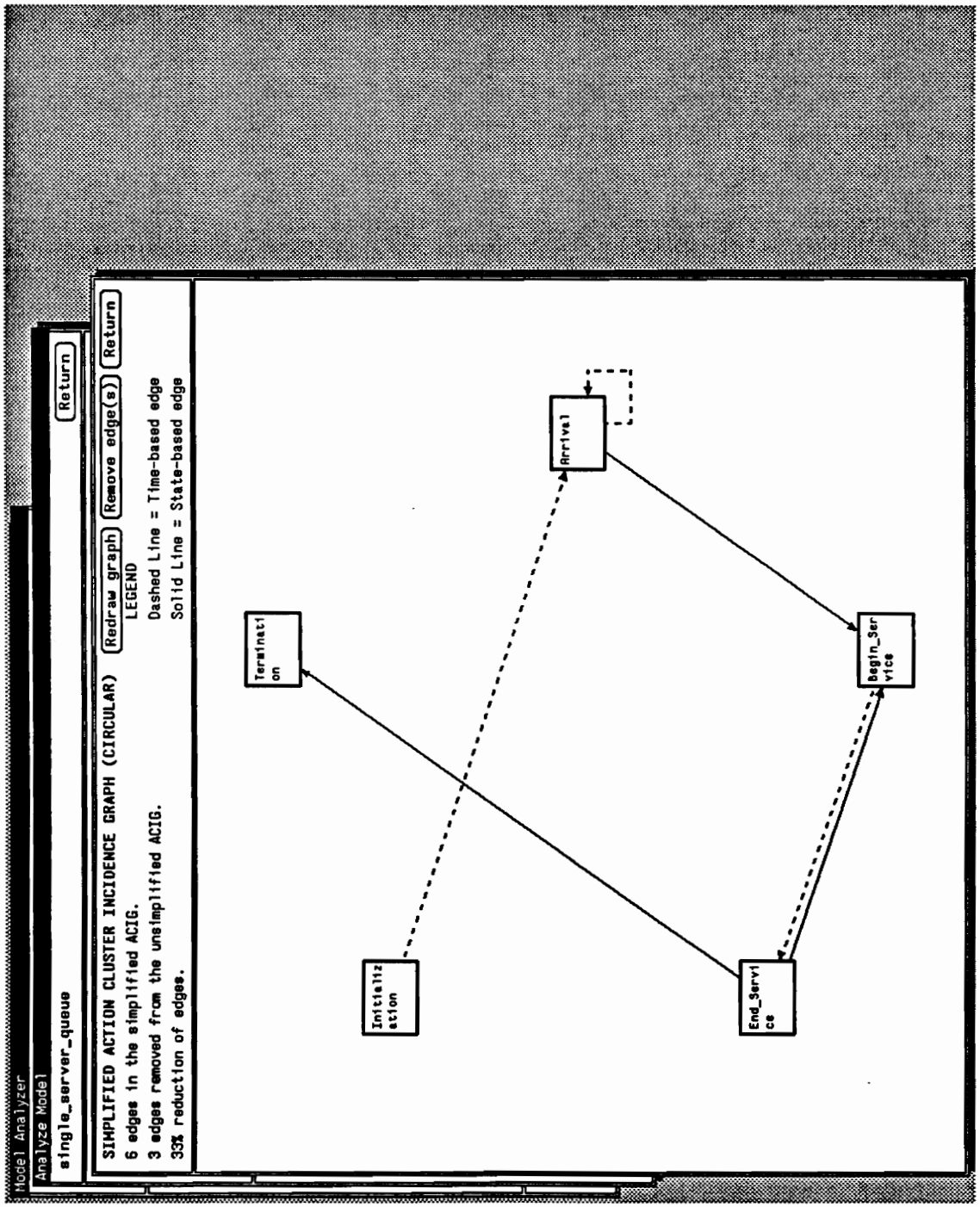


Figure 45. Circular Form of Simplified Action Cluster Incidence Graph

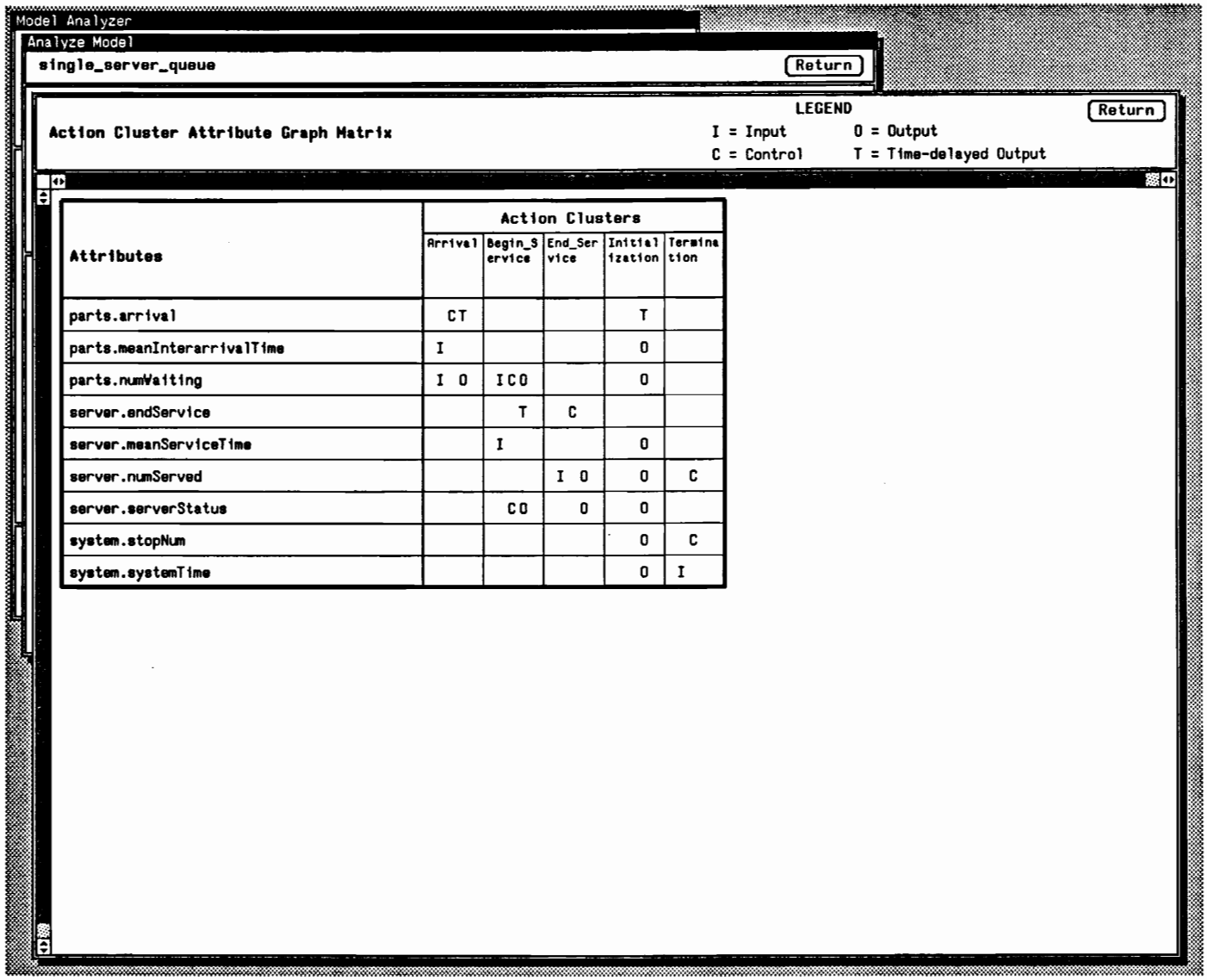


Figure 46. Matrix Form of Action Cluster Attribute Graph

matrix, which means the attribute "parts.arrival" is used as a control and time-delayed output attribute within the "Arrival" action cluster.

4.5.2.3 Attribute Interaction Matrix

The attribute interaction matrix, described in Section 3.1.1, is a Boolean resultant matrix that displays the potential for model attributes to influence the values of other model attributes. Selecting the "View AIM" button in the Model Analyzer displays the AIM for a given model, as shown for the Single Server Queue Model in Figure 47. Each row and column in the matrix represents a model attribute. An *X* in the (*i*, *j*) position signifies that attribute *i* has the potential to influence the value of attribute *j* directly. To influence the value of attribute *j* directly, attribute *i* must be used as a control or input attribute for the same action cluster having attribute *j* as an output attribute.

The AIM is a helpful visual technique of determining the interactions among model attributes. For example, Figure 47 shows attribute "parts.endService" has the potential to affect the values of the attributes "server.numServed" and "server.serverStatus" directly. More importantly, the AIM also determines which interactions between attributes are never possible. In Figure 47, the attributes, "system.stopNum" and "system.systemTime", do not affect the value of any other attribute, nor does any attribute affect the value of "system.stopNum" or "system.systemTime". The values of "parts.meanInterarrivalTime" and "server.meanServiceTime" are also not affected by any attributes.

4.5.2.4 Action Cluster Interaction Matrix

The action cluster interaction matrix, like the AIM, is a boolean resultant matrix but depicts the interactions among action clusters instead of attributes. To view the

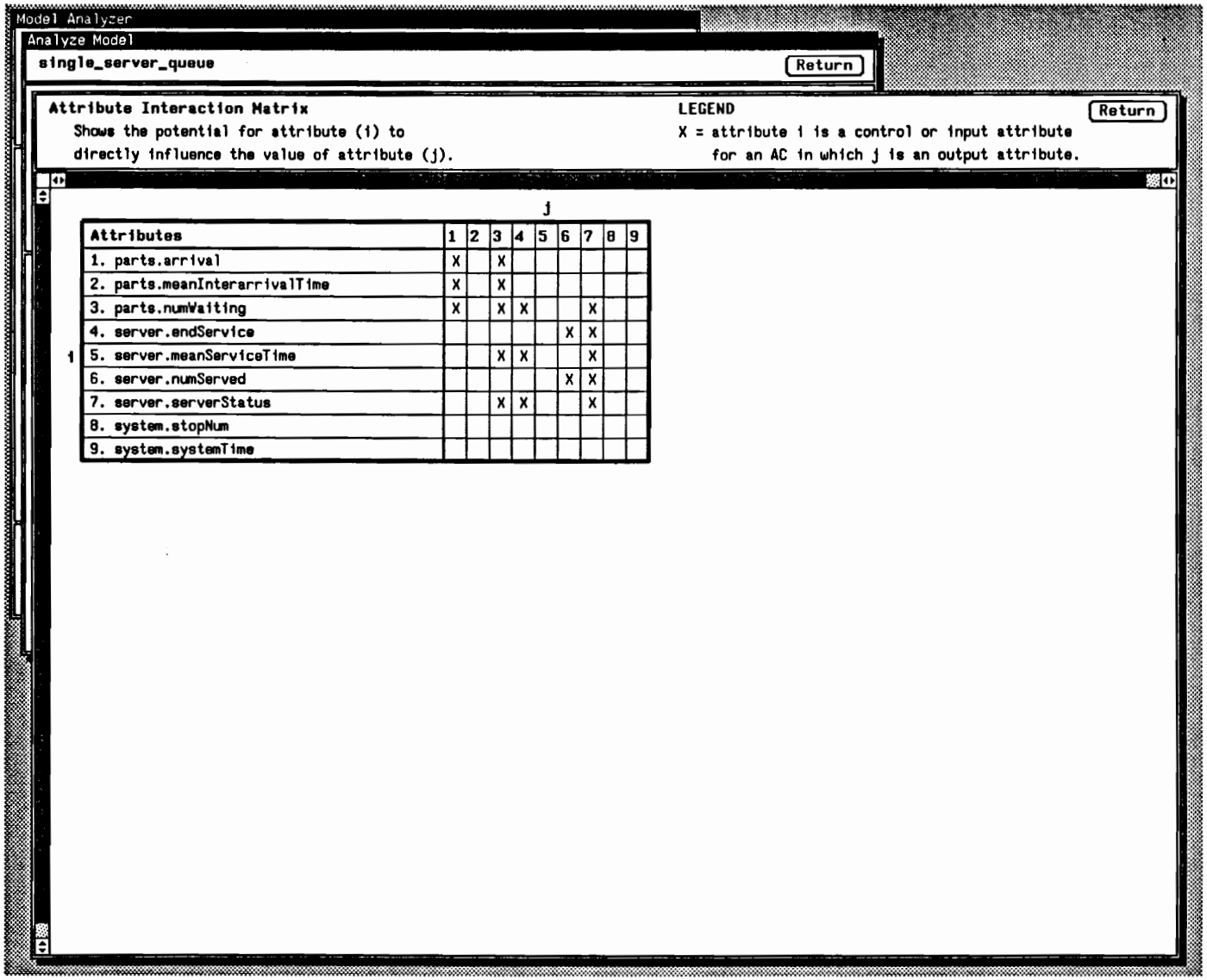


Figure 47. Single Server Queue Attribute Interaction Matrix

ACIM in the Model Analyzer, the "View ACIM" button is selected. The rows and columns of the ACIM, shown in Figure 48, represent the action clusters of the model. An *X* in the (*i*, *j*) position of the matrix represents the ability of AC_i to influence AC_j directly. An action cluster can influence another action cluster directly only if an attribute in one action cluster can affect the value of an attribute in the other action cluster. Stated another way, AC_i can influence AC_j, if AC_i has an output attribute that is an input or control attribute of AC_j.

From this visual display of the ACIM in Figure 48, the "Arrival" AC can influence itself and the "Begin_Service" AC. Another interesting observation shows that the "Termination" AC cannot influence any AC, and the "Initialization" AC cannot be influenced by any AC.

4.5.3 Diagnostic Techniques

The diagnostic techniques performed by the Model Analyzer are listed in the lower right portion of the Analyze Model panel, shown in Figure 49. These techniques are classified into three categories of diagnosis assistance, described in Section 3.1.3 and Table 5.

To perform a particular diagnostic upon the CS of a model, the modeler places a check mark in the box beside the desired diagnostic with a click of the mouse button and then selects the "Start Analysis" button. The modeler also may perform more than one diagnostic technique simultaneously by checking any number of boxes. Figure 49 shows check marks in the boxes corresponding to *attribute utilization* and *attribute initialization*. When the "Start Analysis" button is selected, the Model Analyzer performs all the diagnostic techniques that have a check mark in the box next to them, compiles the results in a file, and displays the file as a window on the screen.

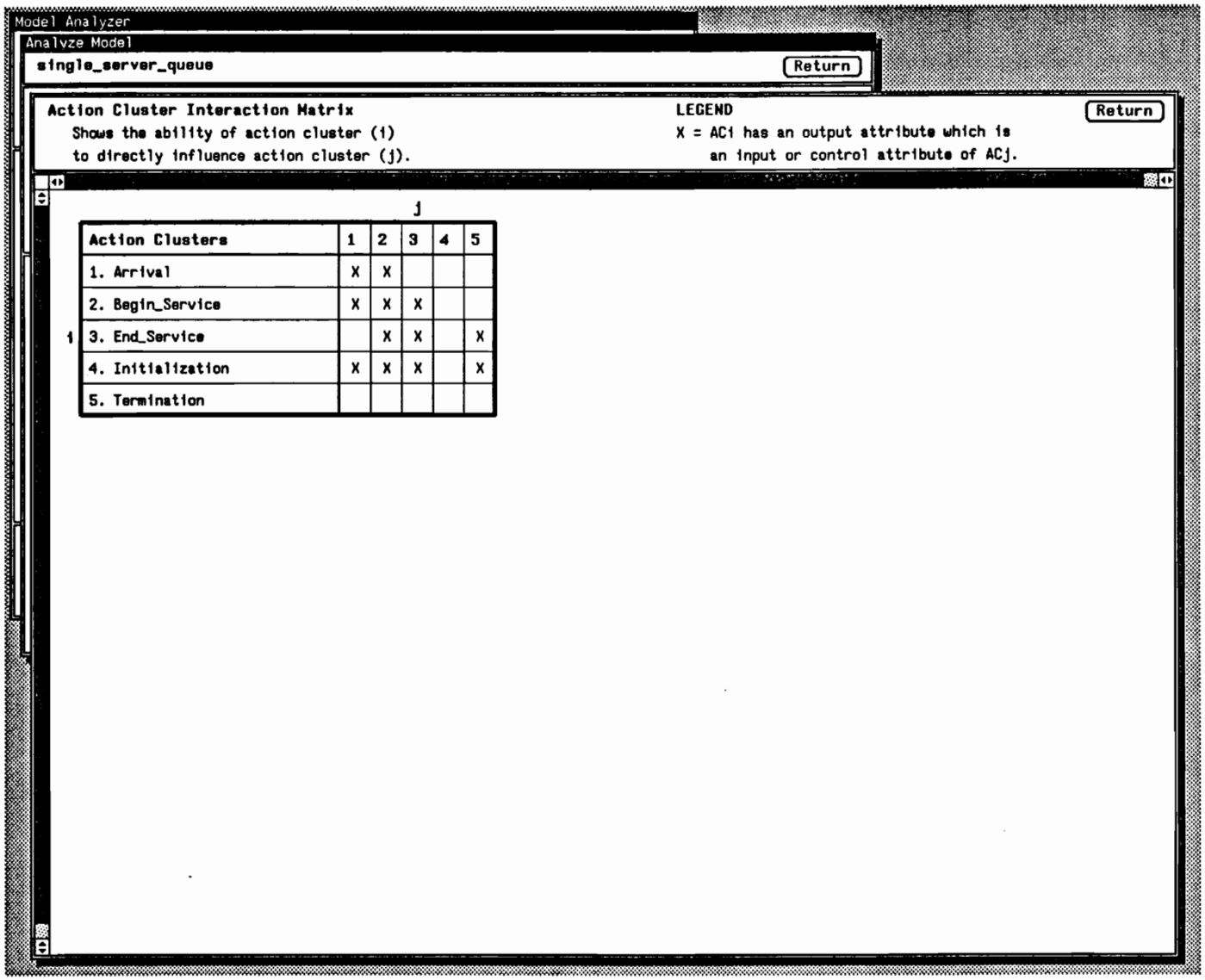


Figure 48. Single Server Queue Action Cluster Interaction Matrix

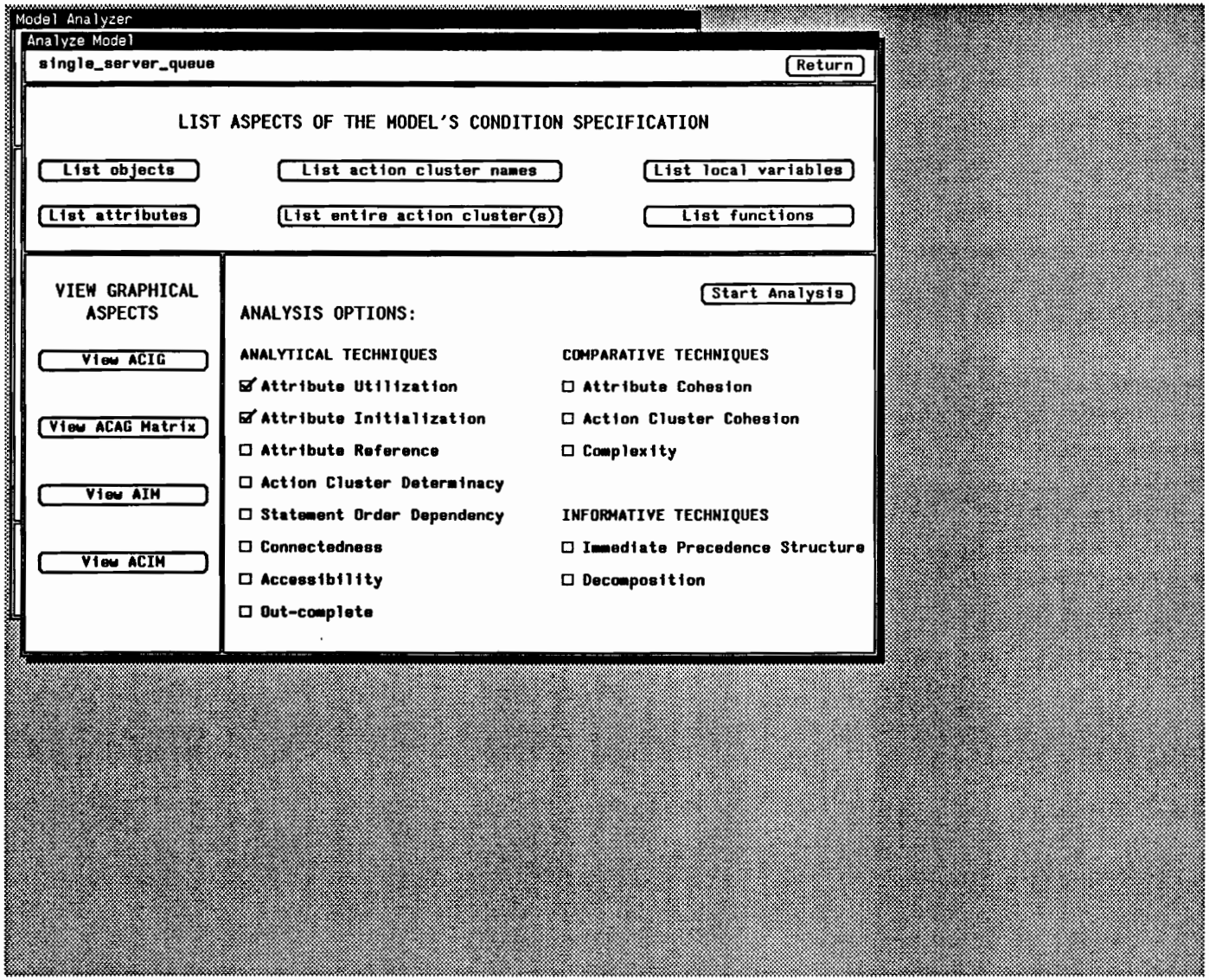


Figure 49. Two Selected Diagnostic Techniques in the Model Analyzer

The analytical diagnostic techniques performed in the Model Analyzer are described in Section 4.5.3.1. Section 4.5.3.2 explains the comparative techniques, and the informative techniques of the Model Analyzer are examined in Section 4.5.3.3.

4.5.3.1 Analytical Diagnostics

Analytical diagnostics determine if the model representation possesses given properties. Of the three types of diagnostic analysis, analytical techniques are the easiest to automate. The Model Analyzer can perform eight different analytical techniques. Comparing the analytical techniques to those listed in Table 5, the Model Analyzer performs six of the eight techniques. *Attribute consistency* and *revision consistency* are not implemented in the Model Analyzer; however, two new analytical techniques have been defined: *attribute reference* and *statement order dependency*. The analytical diagnostic techniques of the Model Analyzer are described in the following eight sections.

4.5.3.1.1 Attribute Utilization

Attribute utilization, described in [Nance and Overstreet 1986, p. 16; 1987a, p. 46; 1987b, p. 593], determines if any attributes are defined but not utilized in the specification of the model. An attribute is utilized if it affects the value of at least one other attribute. Unutilized attributes cannot influence model behavior and may be deleted from the model unless they are being used for statistical or reporting purposes.

The Model Analyzer classifies an attribute utilized if it is used as a control or input attribute somewhere in the CS for that model. Any unutilized attributes are listed by the analyzer. Figure 50 shows the results of attribute utilization for the Single Server Queue Model.

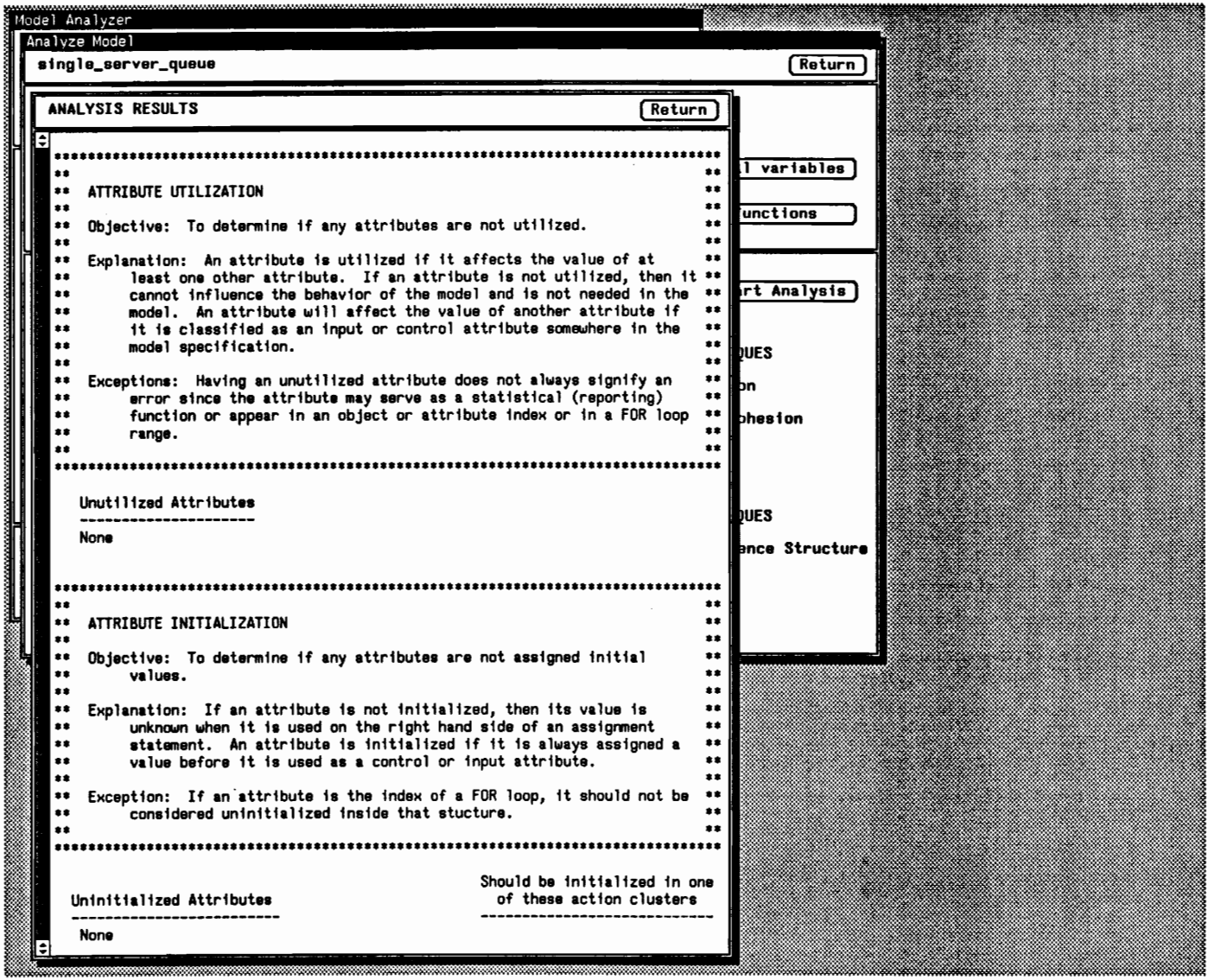


Figure 50. Single Server Queue Attribute Utilization and Attribute Initialization Results

4.5.3.1.2 Attribute Initialization

Another analytical diagnostic performed by the Model Analyzer is *attribute initialization*, which is described in [Nance and Overstreet 1986, p. 16; 1987a, p. 46]. This diagnostic determines if any attributes are not assigned initial values before they are used in the model.

Nance and Overstreet [1986, p. 16; 1987a, p.46] determine an attribute uninitialized, if the attribute has an in-degree of zero in the ACAG, which means the attribute is never used as an output attribute. Satisfying this condition guarantees an uninitialized attribute, because an attribute is only initialized when used as an output attribute, but other attributes used as output attributes also may be uninitialized. During execution of the model, if an attribute is used as a control or input attribute before being used as an output attribute, then the attribute is first used without being assigned a value.

The Model Analyzer performs a complete test for uninitialized attributes. First, the analyzer determines the execution of the model by listing all the possible non-looping paths in the simplified ACIG beginning with the Initialization AC. For each input and control attribute of each AC in the model, the analyzer checks if the attribute is used as an output attribute in the same or any preceding AC for every possible path in the simplified ACIG. If one path is found in which the attribute is not used as an output attribute, then the attribute is considered uninitialized. The Model Analyzer displays all uninitialized attributes in the model and then gives a list of possible action clusters in which the attribute should be initialized, as shown in Figure 50 for the Single Server Queue Model.

Since the analyzer tests all possible paths in the graph, the ACIG should be simplified before this diagnostic is attempted. If the ACIG is not simplified, the test takes much longer to complete and may produce inaccurate results.

4.5.3.1.3 Attribute Reference

Attribute reference is a simple analytical technique, which is not listed in Table 5, that determines if any defined attributes are not referenced in the CS. If an attribute is defined in the object specification of the CS, then it should be referenced somewhere in the transition specification.

The Model Analyzer determines a defined attribute unreferenced if it is not used as a control, input, or output attribute in the model. Figure 51 displays the unreferenced attributes for the Single Server Queue Model.

4.5.3.1.4 Action Cluster Determinacy

Action cluster determinacy, which is labeled *action cluster completeness* in Table 5 and described in [Nance and Overstreet 1986, p. 16; 1987a, p. 46; 1987b, p. 593], is an analytical technique that decides if the required state changes are possible within each action cluster. The actions of an action cluster must cause its condition to become false to prevent an "infinite loop" situation.

This test is performed by the Model Analyzer by checking if an action cluster contains any state-based control attributes, then at least one of these attributes must be an output attribute of the same action cluster. If this check fails, then the action cluster is indeterminate and an "infinite loop" may occur. The indeterminate action clusters for the Single Server Queue Model are listed in Figure 51.

4.5.3.1.5 Statement Order Dependency

Another analytical technique not defined in Table 5, *statement order dependency*, determines if the ordering of the actions (statements) within an action cluster are important. If an action cluster contains one statement that uses an attribute as an input

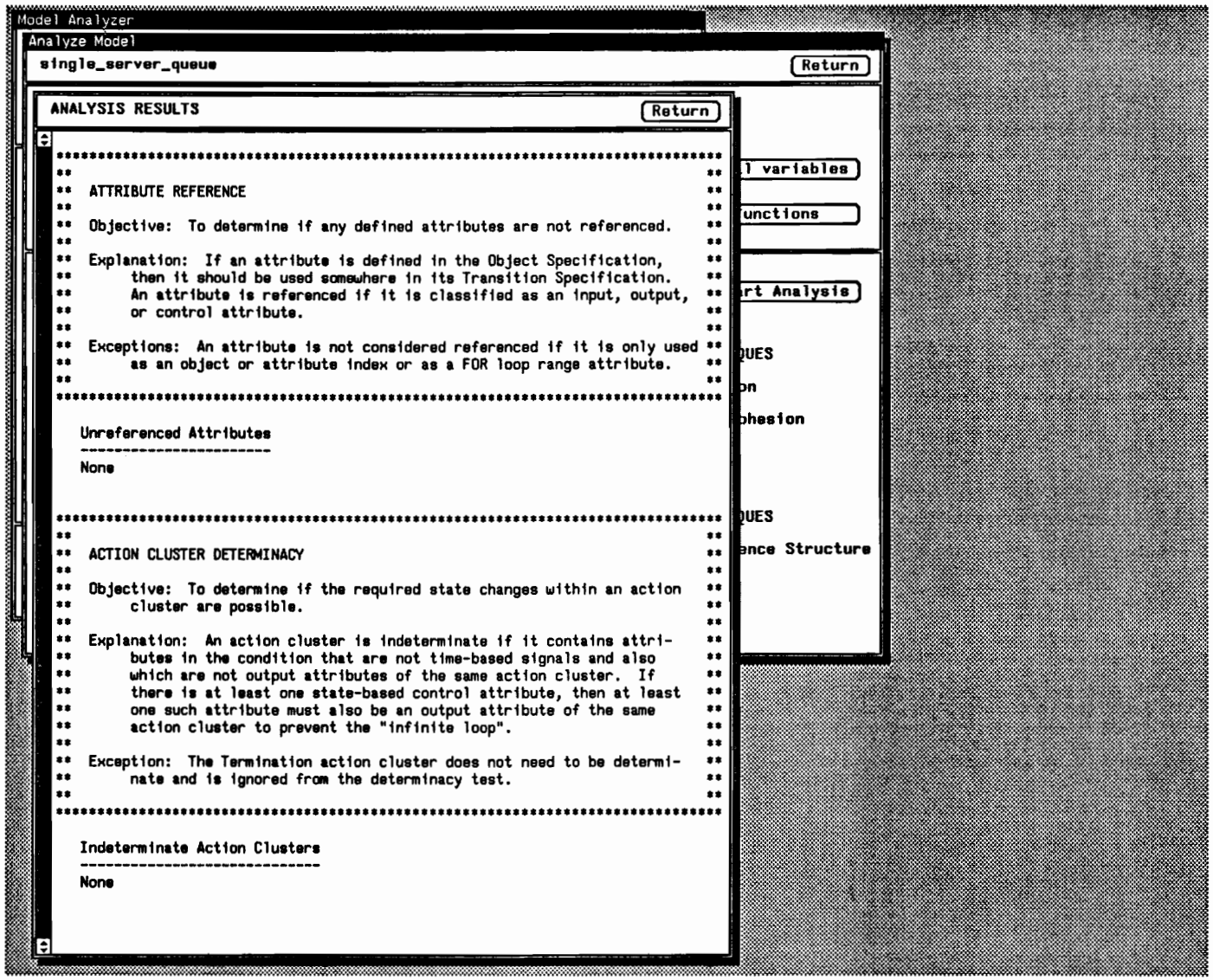


Figure 51. Single Server Queue Attribute Reference and Action Cluster Determinacy Results

attribute and another statement uses the same attribute as an output attribute, then the modeler must manually specify the order of the two statements within the action cluster.

If an attribute is used as an input and output attribute within the same action cluster, then the statements within the action cluster are order dependent. The Model Analyzer lists each order dependent action cluster and its attributes that are used as input and output attributes, as shown for the Single Server Queue Model in Figure 52.

When the same attribute is used as an input and output attribute in the same statement, the action cluster should not be determined as order dependent. For example, the "Arrival" AC in the Single Server Queue Model contains the statement:

parts.numWaiting := parts.numWaiting + 1;

The attribute *parts.numWaiting* is used as an input and output attribute, but since it is used in the same statement, the "Arrival" AC is not order dependent. The implementation of this technique does not determine if the attributes are contained in the same statement and lists those ACs as order dependent.

4.5.3.1.6 Connectedness

Connectedness, an analytical technique described in [Nance and Overstreet 1986, p. 21; 1987a, p. 48], determines if the ACIG is fully connected. Excluding the Initialization AC and its edges, the ACIG is connected if no action clusters are isolated in the graph. If the ACIG is not connected, then the specification is composed of submodels that cannot influence each other. Having distinct submodels within a model is not necessarily an error, but knowledge of this property is helpful to the modeler.

Ignoring the Initialization AC and its edges, the Model Analyzer performs this test by selecting an arbitrary AC and determining its connecting ACs. If all the ACs in the simplified ACIG are connected to the arbitrary AC, then the ACIG is connected, else

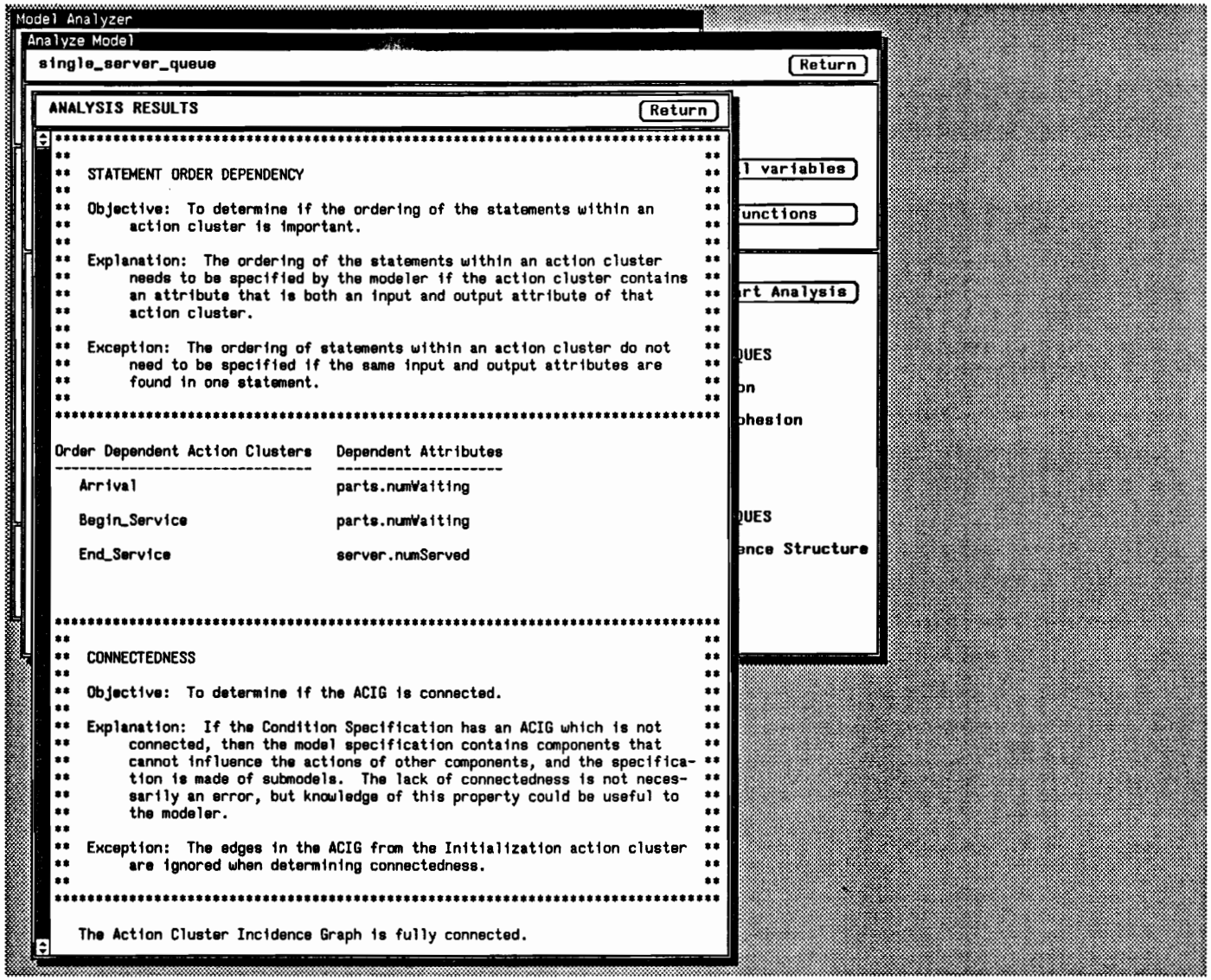


Figure 52. Single Server Queue Statement Order Dependency and Connectedness Results

the group of ACs is listed as a submodel, and the test continues to find additional submodels in the simplified ACIG. Figure 52 lists the results of the *connectedness* test for the Single Server Queue Model.

4.5.3.1.7 Accessibility

Accessibility is an analytical technique that determines if all action clusters are accessible from the Initialization AC. If the ACIG is not completely accessible, then it contains actions clusters that can never be activated in any subsequent execution. Nance and Overstreet [1986, p. 21; 1987a, p. 50] define the CS accessible if the ACIG contains no nodes with an in-degree of zero (other than the Initialization node). This definition does not cover the possibility of an ACIG that has a group of ACs that are fully connected to each other, but does not contain an edge from any of the other ACs, which are connected to the Initialization AC. Here, every AC in the ACIG has an in-coming edge, but a path is not found from the Initialization AC to every other AC in the graph.

The *accessibility* test is implemented in the Model Analyzer by finding the Initialization AC and traversing the simplified ACIG until all paths are exhausted or each AC has been touched. If each AC is touched, then the ACIG is completely accessible from the Initialization action cluster, else the ACIG is not completely accessible and the inaccessible ACs are listed. Figure 53 shows the *accessibility* test for the Single Server Queue Model.

4.5.3.1.8 Out-Completeness

The analytical technique of *out-completeness*, defined in [Nance and Overstreet 1986, p. 21; 1987a, p. 50], determines that no action clusters are specified that cannot influence other action clusters. If the simplified ACIG contains no action clusters with an

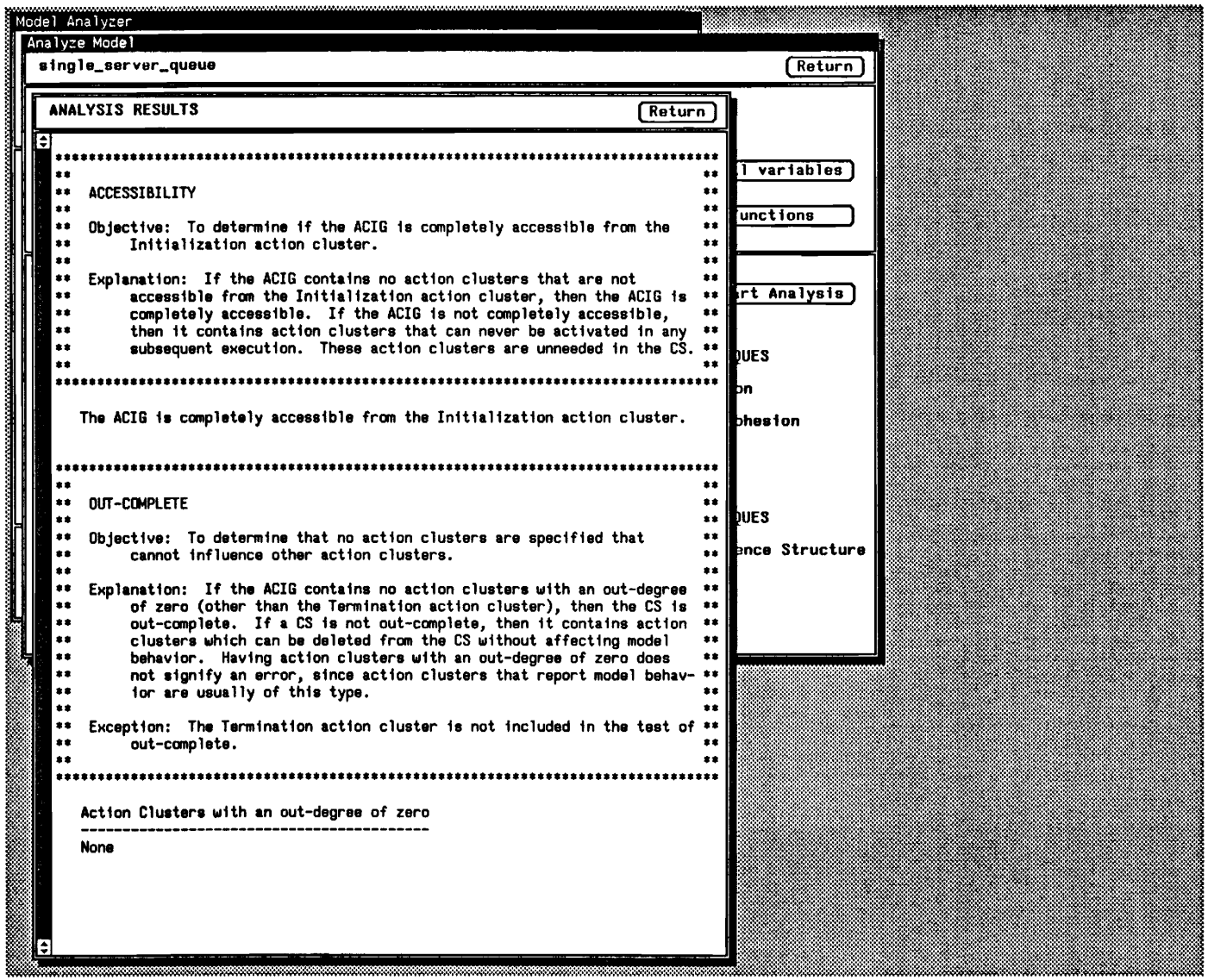


Figure 53. Single Server Queue Accessibility and Out-Completeness Results

out-degree of zero (other than the Termination AC), then the CS is out-complete. If the CS is not out-complete, then it contains action clusters that can be deleted from the CS without affecting model behavior. Having action clusters with an out-degree of zero does not always signify an error, since action clusters that report model behavior are usually of this type.

The Model Analyzer lists each action cluster, except the Termination AC, that does not have any out-going edges. The test for *out-completeness* in the Single Server Queue Model is listed in Figure 53.

4.5.3.2 Comparative Diagnostics

Comparative diagnostics, which are more involved than analytical diagnostics, measure certain characteristics that must be compared with other models to give meaning to the measurements. Each of the three comparative diagnostics defined in Table 5 are implemented in the Model Analyzer, and the following three sections describe these comparative diagnostics.

4.5.3.2.1 Attribute Cohesion

Attribute Cohesion is a comparative diagnostic that determines the degree to which attribute values are mutually influenced and is defined in [Beams 1988, p. 11; Nance and Overstreet 1986, p. 18; 1987a, p. 47]. A low action cluster cohesion index represents a strong relationship between attributes and a high cohesion among model attributes; whereas a higher index represents a more distant relationship between attributes and less cohesion among attributes.

Excluding attributes representing exogenous input, each model attribute can influence the value of all other model attributes. This influence can be direct (attribute *i*

is used in the expression for computing the value of attribute j) or indirect (attribute *i* is used in the expression for computing the value of attribute *j*, which appears in the expression for attribute *k*). The attribute cohesion index represents the minimum number of levels of indirection needed for each attribute to be able to influence the value of all other attributes.

The attribute interaction matrix (AIM) indicates the potential for one attribute to influence another attribute directly. Summing each power of the AIM until the matrix is full represents the level of indirection in which all attributes can influence the value of every other non-exogenous attribute.

The Model Analyzer computes the attribute cohesion index in the following manner. First, it takes the first power of the AIM, which represents all the paths in the AIM graph of length one (attribute *i*influences attribute *j*). Multiplying the AIM by itself will get the second power of AIM, which represents all the paths of length two in the AIM graph (attribute *i* influences attribute *j*, which influences attribute *k*). Summing the two matrices creates a matrix that represents all paths in the graph of length one or two. If the matrix is full (each node can reach every other node by a path of length one or two), then the attribute cohesion matrix is two, else the test continues by finding the matrix that represents all paths of length three or less. This continues until each node in the AIM graph can reach every other node within in a given length, which then becomes the attribute cohesion index. Figure 54 shows the attribute cohesion index for the Single Server Queue Model.

4.5.3.2.2 Action Cluster Cohesion

A similar comparative technique to attribute cohesion is *action cluster cohesion*, which determines the degree to which action clusters are mutually influenced. *Action*

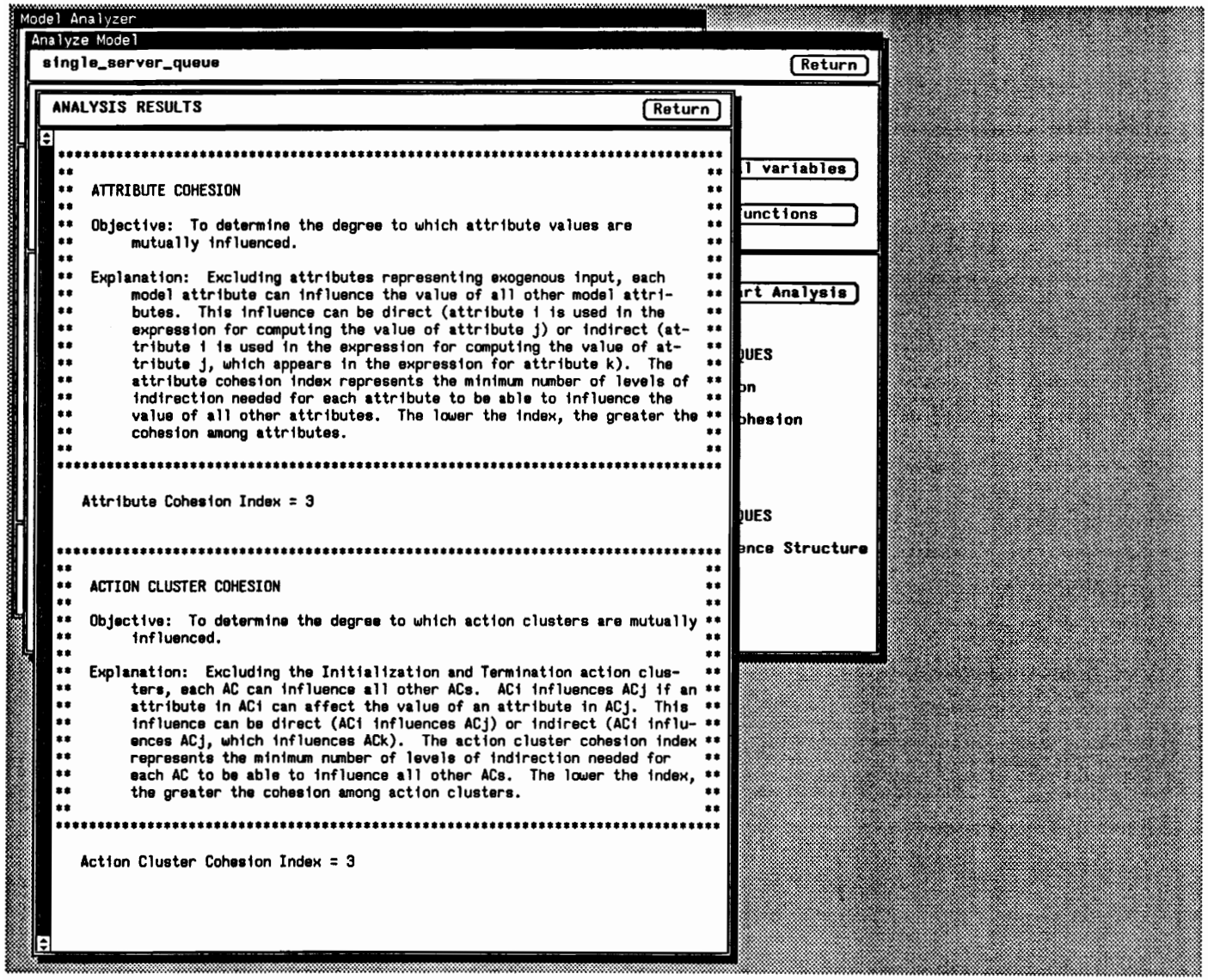


Figure 54. Single Server Queue Attribute Cohesion and Action Cluster Cohesion Results

cluster cohesion is defined in [Beams 1988, p. 12; Nance and Overstreet 1986, p. 18; 1987a, p. 47]. The lower the action cluster cohesion index, the greater the cohesion among action clusters.

Excluding the Initialization and Termination ACs, each AC can influence all other ACs. AC_i influences AC_j if an attribute in AC_i can affect the value of an attribute in AC_j. This influence can be direct or indirect like attribute cohesion. The action cluster index represents the minimum number of levels of indirection needed for each AC to be able to influence all other ACs.

The action cluster interaction matrix (ACIM) indicates the potential for an action cluster to influence another action cluster directly. Summing each power of the ACIM until the matrix is full represents the level of indirection needed for all ACs to influence every other AC (excluding the Initialization and Termination ACs).

The Model Analyzer determines the action cluster cohesion index using the same algorithm as in attribute cohesion, but substituting the ACIM for the AIM. The action cluster cohesion index for the Single Server Queue Model is shown in Figure 54.

4.5.3.2.3 Complexity

Many different types of *complexity* metrics have been used to determine the complexity of models. This prototype employs Wallace's Control and Transformation (CAT) metric to compute the complexity of the CS of a model. A detailed discussion of the CAT metric is found in [Wallace and Nance 1985; Wallace 1985, 1987].

The CAT metric incorporates both the control and transformation complexity. Control complexity measures the complexity of the interactions among action clusters, and transformation complexity measures the complexity of the function (transformation) performed by an action cluster.

The Model Analyzer lists the control and transformation complexities for each action cluster (excluding the Initialization and Termination ACs) and the CAT complexity for the entire model. Figure 55 shows the results of the *complexity* test for the Single Server Queue Model. For accurate complexity results, the ACIG should be simplified before this technique is performed.

4.5.3.3 Informative Diagnostics

The least defined and most difficult to automate, informative diagnostics include model derivations or extracts that could assist in the verification or correctness of a model. The modeler's ability to recognize something informative is needed to apply these diagnostics. The next three sections describe the three informative techniques, which are implemented in the Model Analyzer and defined in Table 5.

4.5.3.3.1 Attribute Classification

The informative technique, *attribute classification*, identifies the function of each attribute in an action cluster as (1) input, (2) output, or (3) control. The Model Analyzer does not have an informative technique listed as *attribute classification*, but the classification of each attribute is shown when the modeler selects the correct menus for the "List attributes" button, as described in Section 4.5.1.2.

4.5.3.3.2 Immediate Precedence Structure

The *immediate precedence structure* diagnostic, defined in [Nance and Overstreet 1986, p. 26; 1987a, p. 54], provides information about the deterministic sequencing of model actions. This test produces the immediate singular predecessor and immediate singular successor, if any, of each action cluster in the simplified ACIG. If AC_i has an

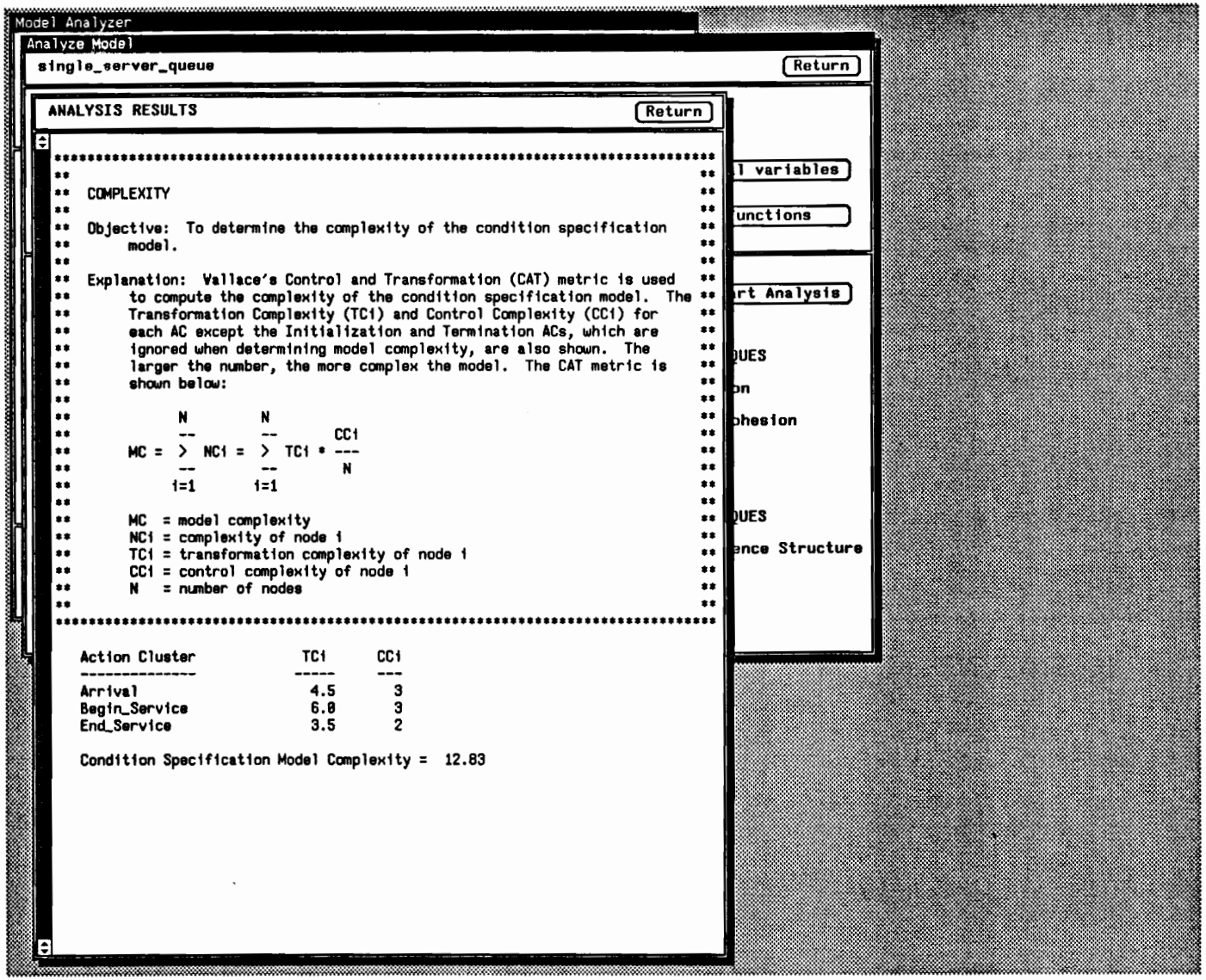


Figure 55. Single Server Queue Complexity Results

in-degree of one, then AC_j with an edge to AC_i is an immediate singular predecessor of AC_i. If AC_i has an out-degree of one, then AC_i has one edge that points to AC_j, which is an immediate singular successor of AC_i. The *immediate precedence structure* test suggests the amount of sequential behavior present in the model, since the immediate singular predecessor of an AC always immediately precedes the execution of the AC, and the immediate singular successor of an AC is always executed after the AC.

The Model Analyzer prints the immediate singular predecessor and successor, if any, of each action cluster. The *immediate precedence structure* results are shown for the Single Server Queue Model in Figure 56. Part of the results show that the "Begin_Service" AC always immediately precedes the execution of "End_Service", and the "Arrival" AC immediately follows the execution of the "Initialization" AC.

4.5.3.3.3 Decomposition

The last informative technique defined in Table 5 and implemented by the Model Analyzer is *decomposition*, which decomposes the ACIG into structures other than action clusters. Usually, the model is broken into subgraphs that consist of related action clusters. The decomposition of the ACIG into subgraphs helps determine the relationships in the model, and may be especially helpful to the modeler during the translation of the model into executable code.

Two previous methods of decomposing the ACIG are outlined by Overstreet [Overstreet 1987; Overstreet and Nance 1986] and Baur, Kochar, and Talavage [1985, 1991]. Overstreet decomposes the ACIG into subgraphs before transforming the CS into one of three discrete event world views: (1) event scheduling, (2) activity scanning, or (3) process interaction. Baur, Kochar, and Talavage decompose the ACIG into

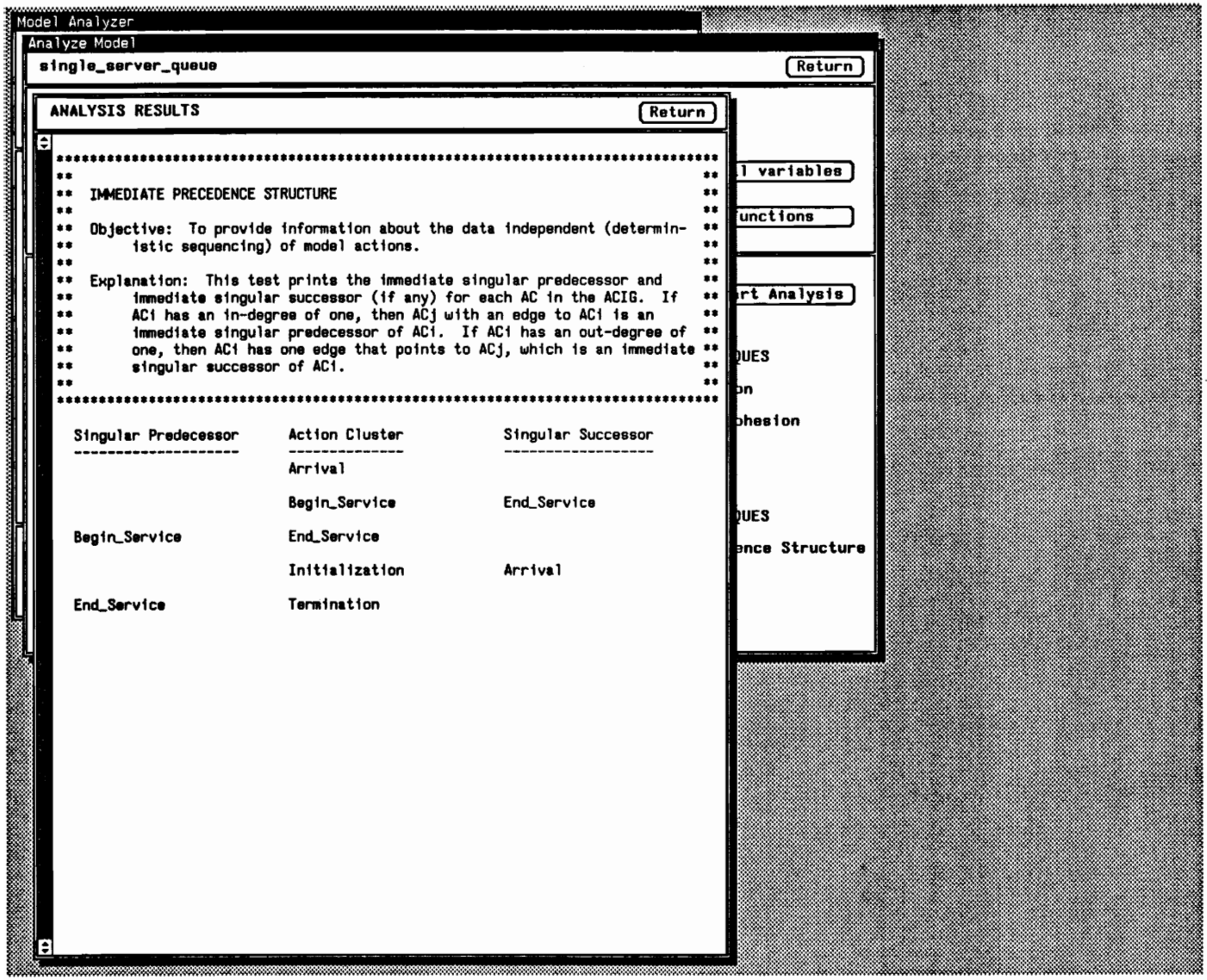


Figure 56. Single Server Queue Immediate Precedence Structure Results

subgraphs by transforming the ACIG into an associated matrix and applying the principal components analysis approach.

The decomposition approach performed by the Model Analyzer is based upon the algorithm for constructing an expanded event graph in [Som and Sargent 1989]. The expanded event graph helps translate the event graph into executable event routines. The decomposition diagnostic in the Model Analyzer modifies the algorithm by Som and Sargent to work on an ACIG instead of an event graph. The goal is to create an expanded ACIG from the simplified ACIG of a model. Throughout this section, the term *ACIG* will always mean *simplified ACIG*. This expanded ACIG helps the modeler translate the CS of a model into an event-scheduling executable model.

The expanded ACIG consists of interconnected subgraphs that have the following properties.

(1) Each subgraph has a unique AC called its *root*. All other ACs in a subgraph are accessible from its root through a path of state-based edges.

(2) Only the root of a subgraph may have an incoming time-delayed edge.

(3) If an edge exists from AC_i of subgraph SG_A to AC_j in subgraph SG_B (SG_A and SG_B are distinct from each other), then AC_j must be the root of SG_B.

(4) If AC_i is a nonroot AC in subgraph SG_A, then no root AC, including the root AC of SG_A, has a higher execution order priority than AC_i.

(5) AC_i is in the expanded ACIG only if AC_i is in the ACIG. (More than one copy of AC_i may exist in the simplified ACIG.)

(6) Two subgraphs cannot have the same root.

(7) In a subgraph, two ACs cannot have the same name (no AC can have more than one copy).

(8) In the expanded ACIG, every copy of AC_i has the same set of outgoing edges as in the original ACIG, and each edge goes to some copy of the same AC_j as in the original ACIG. No other edges exist in the expanded ACIG.

Before the expanded ACIG can be constructed, an *execution order priority matrix*, which defines the order of execution between every pair of ACs in the ACIG, is built. The execution order priority needs to be established between every two distinct action clusters that (1) may occur *simultaneously* and (2) *react* with each other.

Two action clusters, AC_i and AC_j, *react* if $AC_i \neq AC_j$ and one of the following is true.

(1) One AC has an input attribute that is an output attribute in the other AC ($I_i \cap O_j \neq \varnothing$ or $I_j \cap O_i \neq \varnothing$).

(2) One AC has a state-based control attribute that is an output attribute in the other AC ($SC_i \cap O_j \neq \varnothing$ or $SC_j \cap O_i \neq \varnothing$).

(3) One AC has an output attribute that is an output attribute in the other AC ($O_i \cap O_j \neq \varnothing$).

Two action clusters *cannot* occur *simultaneously* if one AC is a state-based singular predecessor of the other AC. Also, the Initialization AC may never occur simultaneously with any other action cluster.

The Model Analyzer creates the execution order priority matrix by asking the modeler to select the execution order priority between every two ACs that react and may occur simultaneously. An alert panel, shown in Figure 57, is displayed for each pair of ACs that react and may occur simultaneously, and the modeler manually selects the action cluster that has priority over the other, if priority is needed.

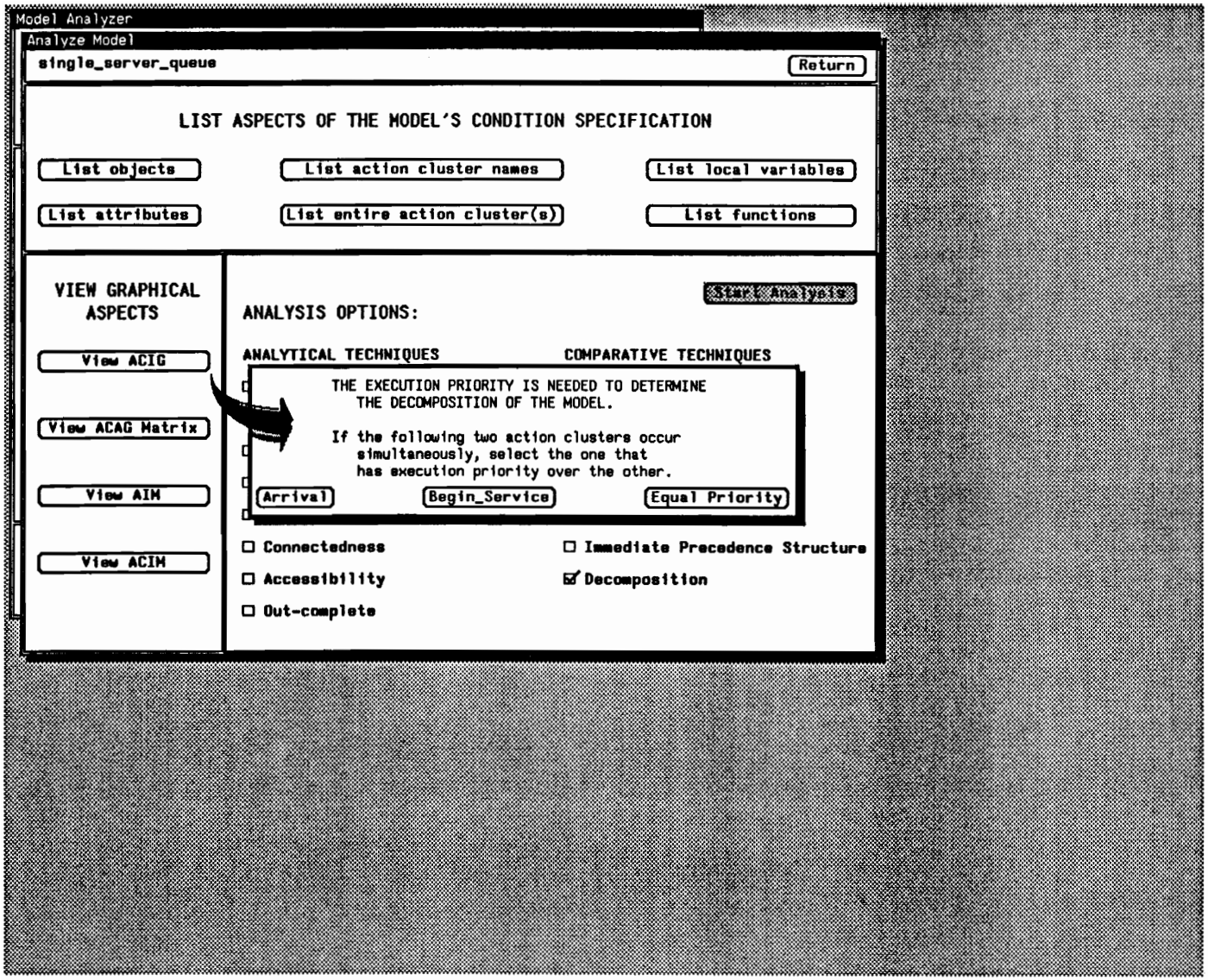


Figure 57. An Alert Panel Used to Determine the Execution Order Priority

After the execution order priority matrix is built, the expanded ACIG is constructed by following a six step algorithm. The first step (Figure 58) identifies the root ACs in the expanded ACIG. In Step 2 (Figure 59), all non-root ACs reachable from a root AC without time-delay and priority conflict are identified for each root AC. This establishes the list of ACs in each subgraph. The next step (Figure 60) develops the secondary root set and subgraphs for each root in the set. The mergeable root set is determined in Step 4 (Figure 61). Some subgraphs determined in the previous step are not really necessary and can be merged with other subgraphs, and the mergeable root set identifies the roots of these subgraphs. In Step 5 (Figure 62), all reachable subgraphs of a root subgraph are merged into that root subgraph. The last step (Figure 63) finds all executable subgraphs of a root subgraph and merges them into the root subgraph.

The results of the algorithm produce a list of subgraphs labeled by their corresponding root AC. Figure 64 shows the decomposition of the Single Server Queue Model. The execution order priorities between action clusters selected by the modeler are also shown. Each subgraph is labeled with its root AC, and the entire set of subgraphs compose the expanded ACIG.

The graphical representation of the expanded ACIG for the Single Server Queue, shown in Figure 65, illustrates how the edges connect each subgraph and the ACs within each subgraph. This graph provides information about the relationships among action clusters in the model. The expanded ACIG also helps translate the CS into an executable model. Representing each subgraph as a *super AC*, the *super Condition Specification* is formed, as shown in Figure 66 for the Single Server Queue Model. A super AC is the combination of the ACs of each subgraph into one large AC. The conditions of non-root ACs are transformed into "if" statements, and some conditions and actions are simplified. Each super AC may represent a separate procedure in the final translation of the model.

STEP 1.

Primary Root Set ← {A: A is an action cluster in the ACIG and satisfies one of the following:

(i) A has no entering edges, or
(ii) A has at least one entering time-based edge.

Root Set ← Primary Root Set

Figure 58. Step 1 of Expanded ACIG Construction Algorithm

STEP 2.

For every action cluster A in the Primary Root Set construct the action cluster set of the subgraph SG_A whose root is action cluster A in the following way:

(i) Action cluster set of $SG_A \leftarrow \{A\}$

(ii) If there is an action cluster B, which is not already a member of SG_A, in the ACIG such that:
   a) B is not in the Root Set and
   b) No action cluster in the ACIG exists with a higher execution order priority than B and
   c) An action cluster in the action cluster set of SG_A exists such that there is a state-based edge from this action cluster to B in the ACIG.

   Then action cluster set of $SG_A \leftarrow$ action cluster set of $SG_A \cup \{B\}$,
   Else stop.

(iii) Go to Step (ii).

Figure 59. Step 2 of Expanded ACIG Construction Algorithm

STEP 3.

Secondary Root Set ← ∅.

For every action cluster A in Root Set *do*
begin
    For every action cluster B in SG_A *do*
    begin
        For every action cluster C such that there is an edge from B to C in the
            ACIG *do*
        begin
            If C is not found in SG_A and C is not found in Root Set
            begin
                Secondary Root Set ← Secondary Root Set ∪ {C}.
                Root Set ← Root Set ∪ {C}.
                Construct action cluster set of SG_A as in Step 2.
            end
        end
    end
end

Note: The Root Set may be expanded during the evaluation of the inner loop and the outer loop must consider these additional action clusters in the Root Set.

Figure 60. Step 3 of Expanded ACIG Construction Algorithm

STEP 4.

(i) Mergeable Root Set ← ∅.

(ii) New Mergeable Roots ← {A: A is an element of the Secondary Root Set and there is no action cluster B in the Root Set such that action cluster B has a higher execution order priority than A}.
If New Mergeable Roots = ∅, then go to Step 5, else go to (iii).

(iii) Mergeable Root Set ← Mergeable Root Set ∪ New Mergeable Roots.

(iv) Root Set ← Root Set - New Mergeable Roots.

(v) Secondary Root Set ← Secondary Root Set - New Mergeable Roots.

(vi) Go to (ii).

Figure 61. Step 4 of Expanded ACIG Construction Algorithm

Definition: We define a binary relation "*reaches*" on the set of action clusters in the ACIG such that *A reaches B* if and only if either

(a) *all* the following are satisfied:
  (i) A is an element of Root Set $\cup$ Mergeable Root Set and B is an element of Mergeable Root Set,
  (ii) $A \neq B$, and
  (iii) there exists an action cluster C in the action cluster set of SG_A such that there is an edge from action cluster C to action cluster B in the ACIG *or*

(b) A reaches D and D reaches B.

Extension of A = {B: A reaches B}.

STEP 5.

For every action cluster A in Root Set *do*
begin
  Compute extension of A.
  Action cluster set of $SG_A \leftarrow$ action cluster set of $SG_A \cup$
    ($\cup_{B \in \text{Extension of A}}$ action cluster set of SG_B).
end

Figure 62. Step 5 of Expanded ACIG Construction Algorithm

Definition: We define a binary relation "*executes*" on the set of action clusters in the ACIG such that *A executes B* if and only if either

(a) *all* the following are satisfied:
  (i) $A \neq B$,
  (ii) A is an element of Root Set and B is an element of Primary Root Set,
  (iii) there exists an action cluster C in the action cluster set of SG_A such that there is a state-based edge from action cluster C to action cluster B in the ACIG, and
  (iv) there is no other action cluster in Root Set with a higher execution order priority than B *or*

(b) A executes D and D executes B.

Executable from A = {B: A executes B}.

STEP 6.

For every action cluster A in Root Set *do*
begin
  Compute the set Executable from A.
  Action cluster set of $SG_A \leftarrow$ action cluster set of $SG_A \cup$
    $(\cup_{B \in \text{Executable from A}}$ action cluster set of $SG_B)$.
end

Figure 63. Step 6 of Expanded ACIG Construction Algorithm

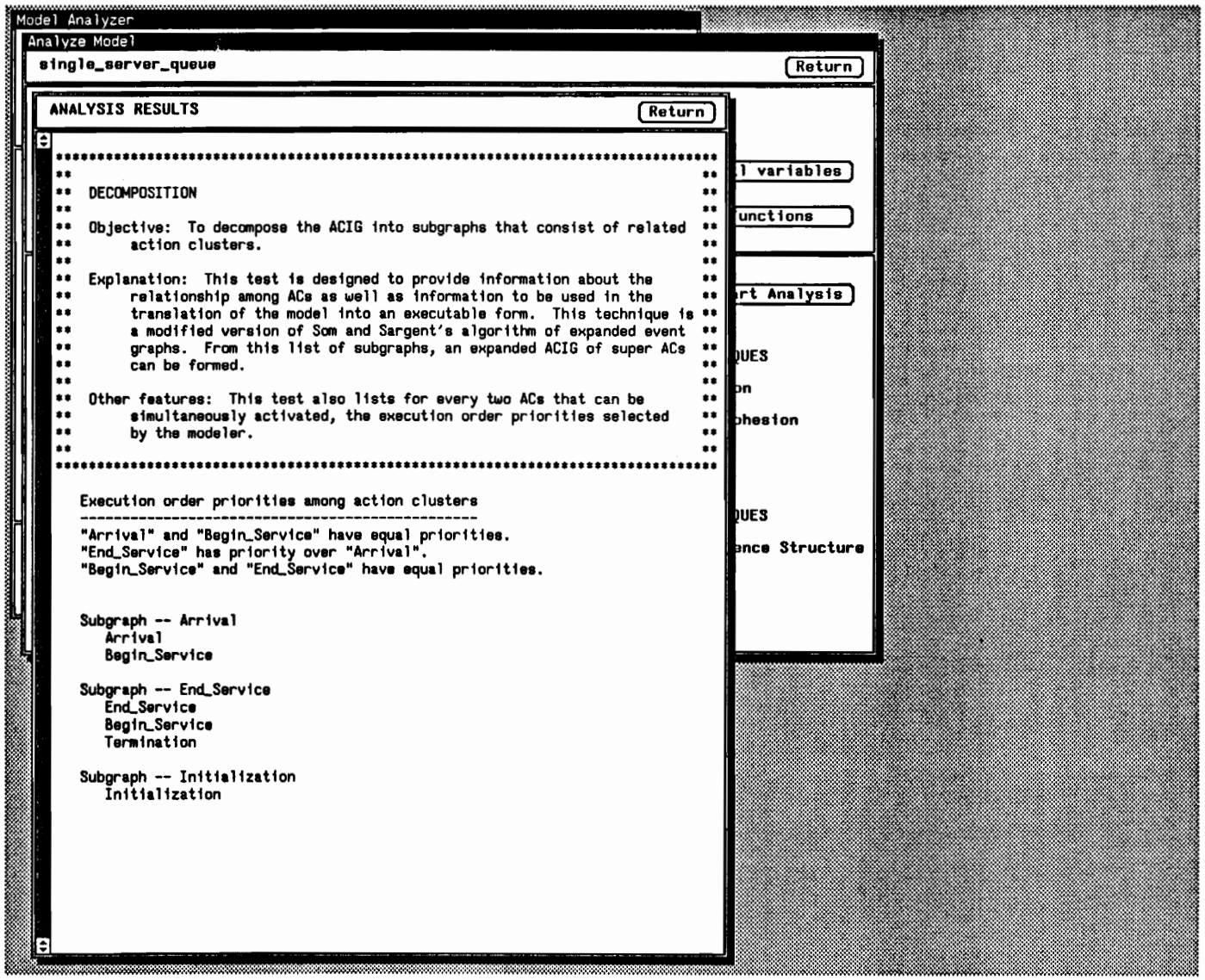


Figure 64. Single Server Queue Decomposition Results

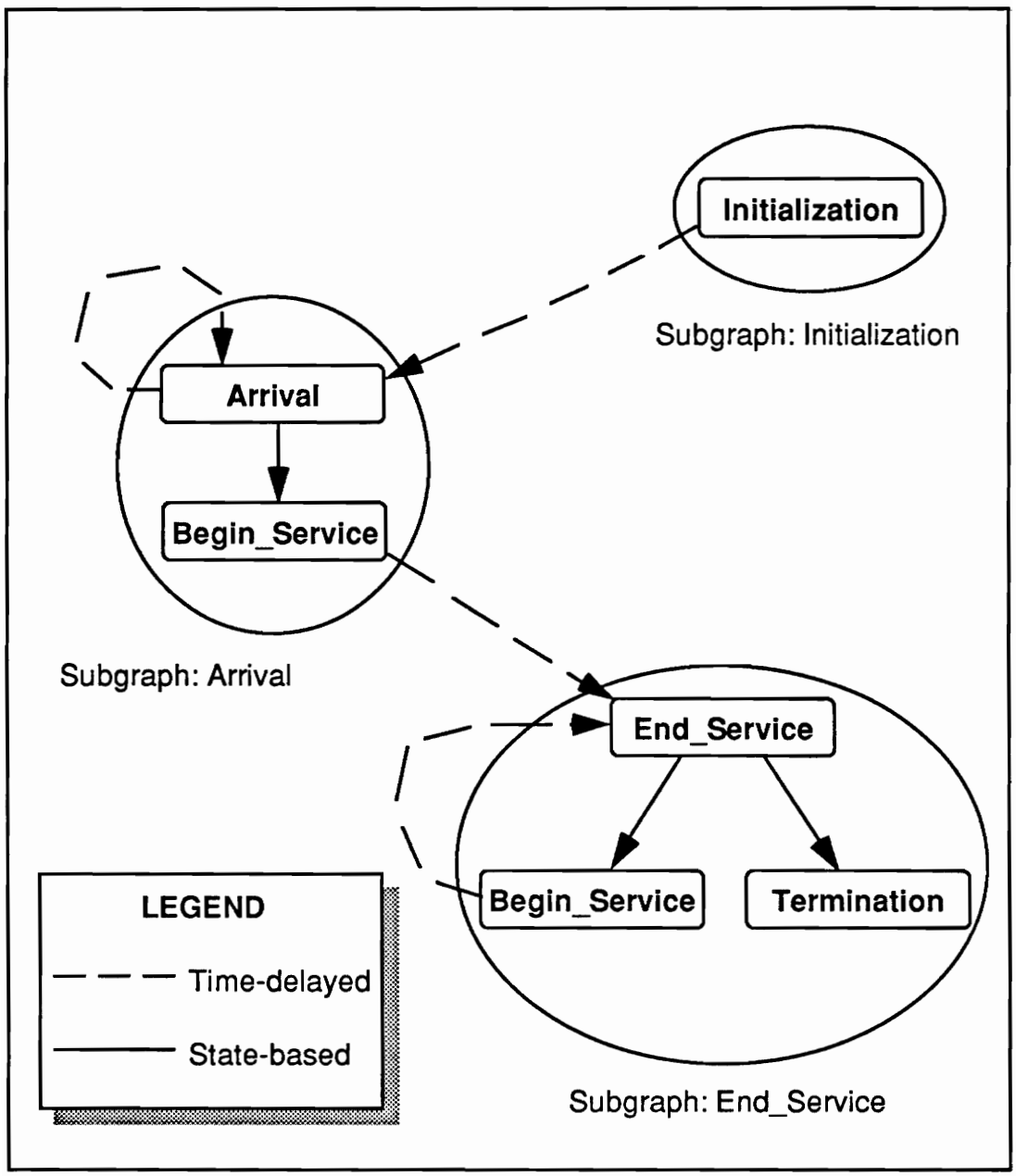


Figure 65. Single Server Queue Expanded Action Cluster Incidence Graph

```
Super AC:  Initialization
      WHEN START:
            system.systemTime := 0;
            server.serverStatus := idle;
            parts.numWaiting := 0;
            server.numServed := 0;
            INPUT (parts.meanInterarrivalTime, server.meanServiceTime,
                  system.stopNum);
            SET ALARM (parts.arrival, 0);

Super AC:  Arrival
      WHEN ALARM (parts.arrival):
            parts.numWaiting := parts.numWaiting + 1;
            SET ALARM (parts.arrival, negexp(parts.meanInterarrivalTime));
            IF server.serverStatus = idle THEN
                  parts.numWaiting := parts.numWaiting - 1;
                  server.serverStatus := busy;
                  SET ALARM (server.endService, negexp(server.meanServiceTime));
            END IF;

Super AC:  End_Service
      WHEN ALARM (server.endService)
            server.numServed := server.numServed + 1;
            server.serverStatus := idle;
            IF parts.numWaiting > 0 THEN
                  parts.numWaiting := parts.numWaiting - 1;
                  server.serverStatus := busy;
                  SET ALARM (server.endService, negexp(server.meanServiceTime));
            END IF;
            IF server.numServed >= system.stopNum THEN
                  OUTPUT (system.systemTime);
                  STOP;
            END IF;
```

Figure 66. Single Server Queue Super Condition Specification

One interesting point about this decomposition procedure is if the execution order priority matrix is ignored and all ACs have equal priorities, the subgraphs of the expanded ACIG are exactly the same as the subgraphs produced by Overstreet and used for the event scheduling world view transformation of the CS. This signifies a relationship between Overstreet's work of translating the CS into an event scheduling world view and Som and Sargent's work of translating an event graph into event routines using an expanded event graph.

5. Action Cluster Incidence Graph Simplification

The current method of producing the action cluster incidence graph from the Condition Specification of a model looks at common attributes among the action clusters. As described in the algorithm in Section 3.1.2, if an output attribute in AC_i is the same as a control attribute in AC_j, then an edge is drawn from AC_i to AC_j. For example, in the First Failed, First Fixed Machine Repairman listed in Appendix C.2, the "Arrive_idle" AC contains the output attribute "repairman.location", which is also a control attribute of the "Travel_to_idle" AC. Therefore, an edge is drawn from the "Arrive_idle" AC to the "Travel_to_idle" AC.

This method of producing the ACIG represents all possible paths among action clusters; however, some created paths are not possible (infeasible). In the example shown above, the "Arrive_idle" AC contains the action "repairman.location := idle"; whereas, part of the condition in the "Travel_to_idle" AC states "repairman.location <> idle". Since "repairman.location" is set to "idle" in the "Arrive_idle" AC, it cannot activate the "Travel_to_idle" AC. Therefore, the edge from "Arrive_idle" to "Travel_to_idle" is infeasible and is not found in the simplified First Failed, First Fixed Machine Repairman ACIG. This additional knowledge, is addition to looking at common attributes among ACs, is needed to produce the simplified ACIG.

Nance and Overstreet [1986, p. 22; 1987a, p.50] state one can further recognize infeasible edges by coupling the expert knowledge of simulation analysis with that of the application domain. This chapter demonstrates, with the help of an expert system, the automatic simplification process of discarding infeasible edges in the ACIG. Since the

simplified ACIG contains fewer edges and is less complicated than the unsimplified ACIG, the modeler can extract more accurate knowledge from the model.

No previous work has been attempted to simplify the ACIG automatically using this expert knowledge. Example ACIGs have been simplified with the help of the modeler, but never automatically. The use of Artificial Intelligence and Expert Systems helps produce this simplification. Overstreet [1982, p. 271] proves an algorithm cannot be found to produce an ACIG that never includes infeasible edges, but this does not stop an expert system from producing an ACIG that rarely includes infeasible edges.

Section 5.1 explains the approach of simplifying the ACIG with the use of an expert system. The simplification techniques used by the expert system are specified in Section 5.2, and Section 5.3 discusses the results of the simplification.

5.1 Expert System Approach

Feigenbaum [Gevarter 1984] defines an *expert system* as an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution. The user interacts with an expert system in a "consultation dialogue", just like interacting with a person with some type of expertise.

One type of expert system is the rule-based expert system, which contains a database of rules, called productions. Each rule is a condition-action pair of the form: IF *condition* THEN *action*, interpreted as "if this condition occurs, then do this action." Using production rules invoked by pattern matching is the most popular approach of representing domain knowledge [Barr and Feigenbaum 1981].

A rule-based expert system, developed in Quintus Prolog on a VAX 8600 (VTCC1), is used to produce a simplified ACIG from the CS of a discrete-event

simulation model. Nance and Overstreet [1986, p. 25; 1987a p. 53] discuss two types of knowledge required for the simplification of an ACIG: (1) knowledge generic to simulation and (2) knowledge specific to the application domain. This expert system encompasses the generic simulation knowledge that will work on models of any application domain. The expert system is not guaranteed to produce a totally simplified ACIG without any infeasible edges, but it does produce ACIGs with fewer edges than using the method described in Section 3.1.2. The expert system is not a part of the Model Analyzer, but the results of the expert system are used by the modeler to simplify the ACIG for a model within the Model Analyzer.

5.1.1 Expert System Inputs

To produce the ACIG, the expert system needs the Condition Specification as its input. The CS must be in a readable Prolog form; therefore, the CS is transformed into a Prolog list structure.

Appendix D lists the CS of three models used by the expert system. The syntax of the CS is similar to the syntax of the CS used by the Model Analyzer, described in Appendix B with CS examples in Appendix C, but is slightly less expressive. The expert system CSs do not use the "dot notation" scheme for representing model attributes and use a FORALL(i) construct instead of the FOR loop used in the Model Analyzer CSs. The CS for the expert system also is limited in its use of multiple instances of attributes and arrays, where the range is not specified and the index must be an *i*.

The CS of a model must be translated into a Prolog list structure before it is input into the expert system. Appendix E displays the Prolog list structures of the three CSs listed in Appendix D. The syntax of the Prolog list structure, outlined in Table 9, is very

Table 9. Syntax of Prolog List Structure of the Condition Specification

Symbol or Keyword	Syntax
Keywords	ALARM | CANCEL_ALARM | CREATE | DESTROY | SET_ALARM | INPUT | OUTPUT | START | STOP | FORALL(i) | FORSOME(i) | INTEGER | REAL | BOOLEAN | SIGNAL | POSITIVE | NONNEGATIVE | NONPOSITIVE | NEGATIVE
Boolean operator keyword	NOT
Arithmetic operators	+ | - | * | / | **
Relational operators	< | > | <= | >= | == | <>
Special symbols	(|) | =
Boolean constant keywords	TRUE | FALSE
Integer constants	(- | ε) (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)*
Real constants	(- | ε) (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)* . (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)* ((E (- | ε) (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)*) | ε)
Identifiers	(a | b | . . . | z) (a | b | . . . | z | 0 | 1 | . . . | 9 | _)*

similar to the syntax of the CSs in Appendix D. Each keyword is written in capital letters, and each word or symbol is separated by commas within a list.

The Prolog list structure is composed of two facts: (1) *object_spec* and (2) *transition_spec*. Table 10 shows the syntax of each fact in pseudo-BNF. The *object_spec* contains a list of objects. Each object is a list of its name and its attributes, and each attribute has a name and a type enclosed in a list. The *transition_spec* is a list of action clusters, in which each action cluster is a list composed of its name, conditions, and actions (statements).

The condition in the Prolog list structure is altered from the original CS. The condition is always written as a two level condition with the outer level representing *AND* conditions, and the inner level representing *OR* conditions. Examples of this condition transformation are listed in Table 11. Notice that the keywords WHEN, AND, and OR do not appear in the Prolog list structure, and the fourth example, "WHEN (A and B) or C", must be rewritten in the equivalent expression "WHEN (A or C) and (B or C)" before being transformed into the Prolog list structure condition format.

5.1.2 Expert System Outputs

The expert system outputs the edges of the action cluster incidence graph for the input CS. The output is text-based, not graphical. Each edge of the ACIG is listed in the following format: "*action_cluster_1* --> *action_cluster_2*", which is interpreted as a directed edge from *action_cluster_1* to *action_cluster_2*. The output ACIGs of three models are listed in Appendix F.

Table 10. Pseudo-BNF of Prolog List Structure Facts

object_spec([<<object>> (, <<object>>)*]).

<<object>> ::= [*object_name*, <<attribute>> (, <<attribute>>)*]

<<attribute>> ::= [*attribute_name*, *attribute_type*]

---

transition_spec([<<action_cluster>> (, <<action_cluster>>)*]).

<<action_cluster>> ::= [*a_c_name*, <<condition>>, <<statements>>]

<<condition>> ::= <<time_condition>>
| <<state_condition>>

<<time_condition>> ::= [['**ALARM**', *alarm_name*] (, <<and_condition>>)*]

<<state_condition>> ::= [<<and_condition>> (, <<and_condition>>)*]

<<and_condition>> ::= [*conditional_phrase*]
| [<<or_condition>> (, <<or_condition>>)*]

<<or_condition>> ::= [*conditional_phrase*]

<<statements>> ::= [*statement*] (, [*statement*])*

Table 11. Examples of Conditions in Prolog List Structure

Condition Specification	Prolog List Structure
WHEN A and B and C	[A, B, C]
WHEN A or B or C	[[A, B, C]]
WHEN (A or B) and C	[[A, B], C]
WHEN (A and B) or C WHEN (A or C) and (B or C)	[[A, C], [B, C]]
WHEN ALARM (*alarm_name*)	[['ALARM', *alarm_name*]]
AFTER ALARM (*alarm_name* and B)	[['ALARM', *alarm_name*], B]
A, B, C ::= [*conditional_phrase*]	

5.2 Simplification Techniques

The expert system uses a conservative approach to produce the simplified ACIG. Edges are deleted from the unsimplified ACIG only if they can be proven infeasible, which prevents edges from being deleted that should not have been deleted. On the other hand, this expert system may not be able to prove a known infeasible edge infeasible, and the infeasible edge will remain in the ACIG.

Section 5.2.1 explains the expert system strategy and approach of simplifying the ACIG. The rule base of the expert system is described in Section 5.2.2, and the limitations of the expert system are listed in Section 5.2.3.

5.2.1 Strategy and Approach

The top level of the expert system defines an edge. An edge exists between AC_i and AC_j if

(1) AC_i and AC_j are valid action cluster names,

(2) at least one statement in AC_i shares a common attribute with the condition of AC_j (at least one output attribute of AC_i is a control attribute of AC_j), and

(3) the edge cannot be proven infeasible.

The rationale of constructing the ACIG goes one step further than the algorithm in Section 3.1.2, which employs only rules one and two. The third rule, which determines the edge infeasible, is the crucial element in producing a more simplified ACIG. This expert system focuses on how to prove a given edge infeasible.

The expert system implements three methods of proving an edge infeasible. An edge from AC_i to AC_j is infeasible if

(1) statements of AC_i cause the condition of AC_j to be always false,

(2) no statements of AC_i can satisfy the condition of AC_j, or

(3) one part of the condition of AC_i that is not affected by the statements of AC_i causes the condition of AC_j to be always false.

The first method, explained in more detail, indicates the statements of AC_i cause the condition of AC_j to be always false if one statement of AC_i causes one part of an *AND* conditional phrase to be always false, or if several statements of AC_i cause each part of an *OR* conditional phrase to be always false. The expert system contains rules, called Prove False Rules, that determine when a statement causes a conditional phrase to be always false.

An example of this first infeasibility method is applied to the First Failed, First Fixed Machine Repairman Condition Specification listed in Appendix D.2. The edge from the "Arrive_idle" AC to the "Travel_to_idle" AC is infeasible because the statement "location = idle" in "Arrive_idle" causes the condition "WHEN FORALL(i), failed(i) AND status == available AND location <> idle" of "Travel_to_idle" to be false. An edge from "Arrive_idle" to "Travel_to_idle" means the actions or statements of "Arrive_idle" has the ability to satisfy the condition of "Travel_to_idle". This is not true in this example since the statement "location = idle" causes the conditional phrase "location <> idle" to be always false.

The second infeasibility method suggests no statement of AC_i satisfies the condition of AC_j if each statement of AC_i that shares a common attribute with the condition of AC_j does not satisfy either one part of an *AND* or *OR* condition of AC_j. The

expert system contains rules, called Not Satisfy Rules, that decide when a statement never satisfies a conditional phrase.

An application of this second method is used in the Harbor Model CS located in Appendix D.3. The edge from the "Move_tug_to_pier" AC to the "Enter" AC is infeasible because the statement of "Enter" that contains a common attribute with the condition of "Move_tug_to_pier" is "num_arr_tugs = num_arr_tugs - 1", which never satisfies the condition "WHEN num_arr_q > 0 AND num_arr_tugs > 0 AND num_free_berths > 0" of "Enter". Satisfying the condition implies the condition is originally false and becomes true after applying the statement. If "num_arr_tugs > 0" is false, then "num_arr_tugs" has a value of zero (0) or less. After applying the statement "num_arr_tugs = num_arr_tugs - 1", which subtracts one (1) from the value of "num_arr_tugs", the value of "num_arr_tugs" is never greater than zero (0) to satisfy the condition "num_arr_tugs > 0".

The most complex infeasibility method is the third one, which has two steps. First, the system determines all parts of the condition of AC_i that do not share any common attributes with the statements of the same AC_i. If the conditional part is a group of *OR* conditional phrases, each phrase must not have any common attributes with its statements. Second, the system takes this list of conditional phrases of AC_i and determines if any of these conditional phrases makes the condition of AC_j always false. If one conditional phrase of AC_i makes one part of an *AND* conditional phrase of AC_j always false, or if several conditional phrases of AC_i make each part of an *OR* conditional phrase always false, then that edge is infeasible. Prove Condition False Rules within the expert system decide when one condition causes another condition to be always false.

One example where this method is applied is in the Harbor Model CS. Here, the edge "Move_tug_to_ocean" to "Move_tug_to_pier" is proven infeasible by the third method because two conditional phrases of "Move_tug_to_ocean" that do not have common attributes in the statements of the same AC, "num_arr_q > 0" and "num_free_berths > 0", cause the condition of "Move_tug_to_pier" to be always false. The *OR* condition of "Move_tug_to_pier", "(num_arr_q == 0 OR num_free_berths == 0)" is always false based upon the two true conditions of "Move_tug_to_ocean"; therefore, the edge is infeasible.

5.2.2 Rule Base of Expert System

The rule base, which is the largest part of the expert system, contains rules that apply the expert knowledge of simulation and decide if an edge is infeasible. The conflict resolution strategies employed are rule ordering and rule separation.

The rules are separated into three types: (1) Prove False Rules, (2) Not Satisfy Rules, and (3) Prove Condition False Rules. Table 12 shows a breakdown of the 791 rules currently in the system. This rule base is not meant to be complete, and as new rules are found to prove an edge infeasible, they will be added to the system.

The first set of rules, Prove False Rules, provides all the possibilities a given statement can make a certain condition false. One example of the 126 Prove False Rules states if the statement is of the form "Att = B" and the condition is in the form "Att < B" where *Att* is a common attribute and *B* is any variable or number, then that statement always makes the condition false.

The Not Satisfy Rules, which form the second set of rules in the system, list the different ways a statement never satisfies a condition. The system contains twenty-five (25) Not Satisfy Rules, but the last rule states that every Prove False Rule is also a Not

Table 12. Rule Base of Expert System

Type of Rule	Number of Rules
Prove False	126
Not Satisfy	25 (150)
Prove Condition False	640
TOTAL	**791**

Satisfy Rule, so 150 Not Satisfy Rules actually exist. One example of a Not Satisfy Rule states if the statement is of the form "Att = Att - <positive_number>" and the condition is of the form "Att > B" where *Att* is a common attribute and *B* is any variable or number, then that statement can never satisfy the condition.

The last set of rules, Prove Condition False Rules, shows the possible ways one condition can cause another condition to be always false. One example of the 640 Prove False Rules is if the first condition is of the form "B > C" and the second condition is of the form "B < C" where *B* and *C* are either variables or numbers, then if the first condition is true, the second condition is always false.

5.2.3 Limitations of Expert System

This expert system, which produces a simplified ACIG, has some limitations. First, the array notation of the Prolog list structure of the CS is very primitive. The only array subscript allowed is *i*, and the keywords *FORALL(i)* and *FORSOME(i)* cannot determine the ranges of *i*.

The second limitation is the ability of the system to define attributes of objects. Each attribute must be unique, the system does not handle the "dot notation" or "object-attribute pair" representation of attributes. For example, both attributes, "repairman1.status" and "repairman2.status", are not allowed in the system.

Another limitation is that common attributes are not found if one is located in an *INPUT* statement, since the current system ignores all attributes of *INPUT* statements. However, the *INPUT* statement is used in the Initialization AC to initialize attributes that are usually system parameters, which are not important to the logic of the model.

Finally, attributes need to be defined in the object specification of the CS as detailed and exact as possible. If an attribute is not exactly defined, some infeasible

edges may not be proven infeasible. For example, "num_repairs" should be defined as *POSITIVE INTEGER* or *NONNEGATIVE INTEGER* instead of just *INTEGER*.

5.3 Results of Simplification

Instructions needed to operate the expert system are contained in Section 5.3.1. Section 5.3.2 discusses the results of three models.

5.3.1 Operation of System

The expert system is implemented in Quintus Prolog on VTCC1. When the prolog system is entered by typing "prolog" at the *$* prompt, the Quintus Prolog prompt, "| ?- ", will appear. First, the Condition Specification Prolog list structure is loaded by typing "['*cs_name*']." where *cs_name* is the file name of the CS Prolog list structure for a model. Next, the expert system is loaded by typing the command "['aiproj.pl']."

The edges of the ACIG are determined by the command *edge*. To find all the edges of the graph, type "edge(AC1, AC2)." where *AC1* and *AC2* are variables that binds to one edge in the ACIG. To find the rest of the edges, type ";" after each answer until the word "no", which stands for no more solutions, appears. To determine if a particular edge is in the ACIG, input "edge('*ac_name1*', '*ac_name2*')." where *ac_name1* and *ac_name2* represent actual action cluster names. For example, to ask if an edge from Initialization to Termination is present, input "edge('Initialization', 'Termination').", and the system will respond with a "yes" or "no".

The output file shown in Appendix F.1 of the Single Server Queue Model is created using the following commands:

```
| ?- ['ssqueue.pl'].
| ?- ['aiproj.pl'].
| ?- tell('ssqueue.out').
```

```
| ?- edge(AC1, AC2), write(AC1), write(' --> '), write(AC2), nl, nl.
(continue to input ";" until the word "no" appears)
| ?- told.
```

The output is written to a file called "ssqueue.out".

5.3.2 Test Results

The expert system is given three different Condition Specifications to evaluate and produce the Action Cluster Incidence Graph. The three examples are the Single Server Queue, the First Failed, First Fixed Machine Repairman, and the Harbor Model problems. The CS for these three models are listed in Appendix D, and Appendix E shows the Prolog list structure of these CSs, which are used as input to the system. The results of each expert system for each model are listed in Appendix F.

In the three test models, the expert system deletes all the infeasible edges, and the edges seen in Appendix F consist of the most simplified ACIGs for each problem. This reduction technique makes a significant reduction in the complexity of each ACIG by removing from one-third to one-half of the edges from the unsimplified ACIG, which is formed by just looking at the common attributes among ACs. Table 13 shows a breakdown of the number and percentage of edges removed in each model. To get a visual appreciation of the reduced complexity of the simplified ACIG, the unsimplified and simplified ACIGs for the three models are shown in Appendix G.

Another important factor besides effectiveness is speed. The speed of simplifying the ACIG depends upon several factors: (1) the number of action clusters, (2) the number of edges in the unsimplified ACIG, (3) and the complexity of each action cluster, which depends upon the complexity of its condition and the number of statements in the AC.

Table 14 shows the length of time to remove the infeasible edges and produce the simplified ACIG for each model. The most complicated model, the Harbor Model, took

Table 13. Effectiveness of Simplification Technique

CS Model	Edges in Unsimplified ACIG	Edges in Simplified ACIG	Edges Removed	% of Edges Removed
Single Server Queue	9	6	3	33%
First Failed, First Fixed Machine Repairman	21	10	11	52%
Harbor Model	45	26	19	42%

Table 14. Comparison of Run Times for Each Model

CS Model	Number of Action Clusters	Edges in Unsimplified ACIG	Edges in Simplified ACIG	Approximate Run Time
Single Server Queue	5	9	6	3 sec.
First Failed, First Fixed Machine Repairman	8	21	10	8 sec.
Harbor Model	12	45	26	30 sec.

roughly thirty (30) seconds for the expert system to remove nineteen (19) of its forty-five (45) edges. Compared to the large number of rules in the expert system, the speed of the simplification process is fast.

This expert system is not designed to totally simplify an ACIG, but to eliminate the common types of infeasible edges. The expert system will improve as its rule set improves. The use of an expert system proves to be an effective tool in producing simplified ACIGs.

6. Example

This chapter demonstrates how the modeler conducts an interactive analysis session with a discrete-event simulation model specification using the Model Analyzer. Many features of the Model Analyzer are shown in this chapter, but a complete description of its operation is explained in the User's Manual in Appendix H.

The chosen model for the analysis session is the popular First Failed, First Fixed Machine Repairman Model. The description of the model follows:

> A single repairman services a group of *n* identical semi-automatic machines. The machines break down at random intervals, based on a Poisson distribution with mean λ. When a machine failure occurs, the repairman travels to the machine's location, repairs the machine, and then travels to the next failed machine's location. If no other machines have failed, the repairman travels back to its idle position. This cycle continues until a specified number of repairs is completed. Service time for a machine follows a negative exponential distribution with mean μ, and a function determines the travel time between any two machines or between the idle position and a machine.

The Condition Specification for the First Failed, First Fixed Machine Repairman is listed in Appendix C.2.

First, the modeler enters the Model Analyzer and selects the First Failed, First Fixed Machine Repairman Model, shown as "fourf_repairman" in Figure 67, from the list of model names. When the "fourf_repairman" is selected, the name appears to the right of the words "Model name:". To determine if the model name selected is the wanted CS model, the modeler views the CS by selecting the eyeball icon next to the words "View Model's Condition Specification". The CS for the First Failed, First Fixed

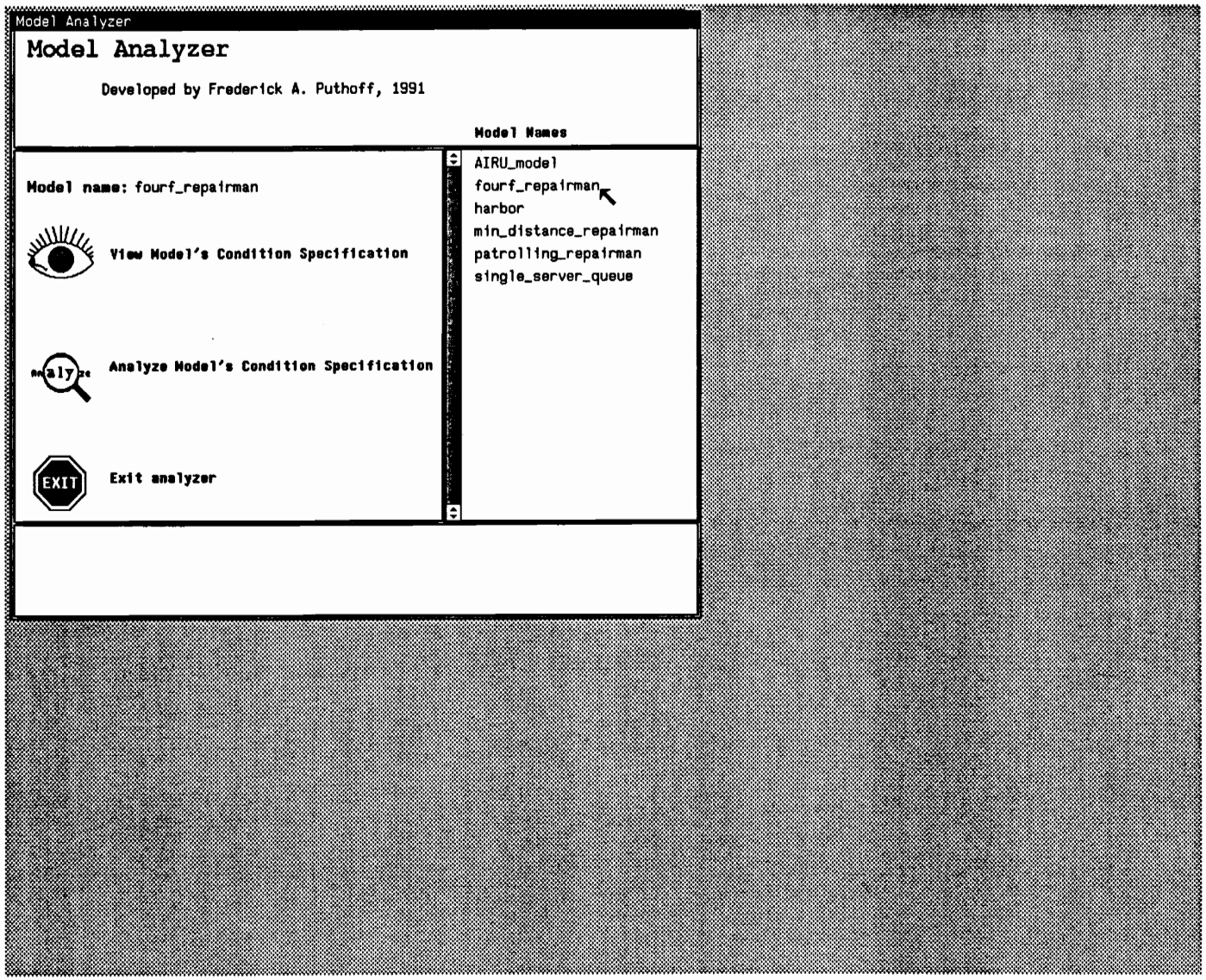


Figure 67. Selecting the First Failed, First Fixed Machine Repairman Model

Machine Repairman Model is then shown on the screen (Figure 68). The modeler uses the scrollbar to see the entire CS, as shown in Figure 69. After finishing viewing the CS, the modeler selects the "Return/Save" button and returns to the main panel.

To analyze the First Failed, First Fixed Machine Repairman CS, the modeler selects the magnifying glass icon next to the words "Analyze Model's Condition Specification" and waits as the Model Analyzer prepares the model before any analysis begins. After a minute or two, the Analyze Model panel appears, illustrated in Figure 70, and the model is ready to be analyzed.

6.1 List Aspects of the Model

The first type of analysis the modeler performs looks at the various parts of the CS to help understand the model and determine if the specification is complete and accurate. The three objects in the First Failed, First Fixed Machine Repairman Model are displayed when the "List objects" button is selected, as seen in Figure 71. To view all the attributes of each object in the model, the "List attributes" button is selected with the corresponding pull-right menu selections of "Of Object(s)", "All Types", and "All Objects". The list that appears, shown in Figure 72, contains all the information defined in the object specification of the CS for the First Failed, First Fixed Machine Repairman Model.

The modeler views a list of the action clusters in the CS by selecting the "List action cluster names" button. The CS contains the eight action clusters shown in Figure 73. To see each complete action cluster, condition and actions, the modeler selects the "Entire action cluster(s)" button and the "All Action Clusters" menu item. The panel, shown in Figure 74, contains all the information found in the transition specification of the First Failed, First Fixed Machine Repairman Model. The classification of each

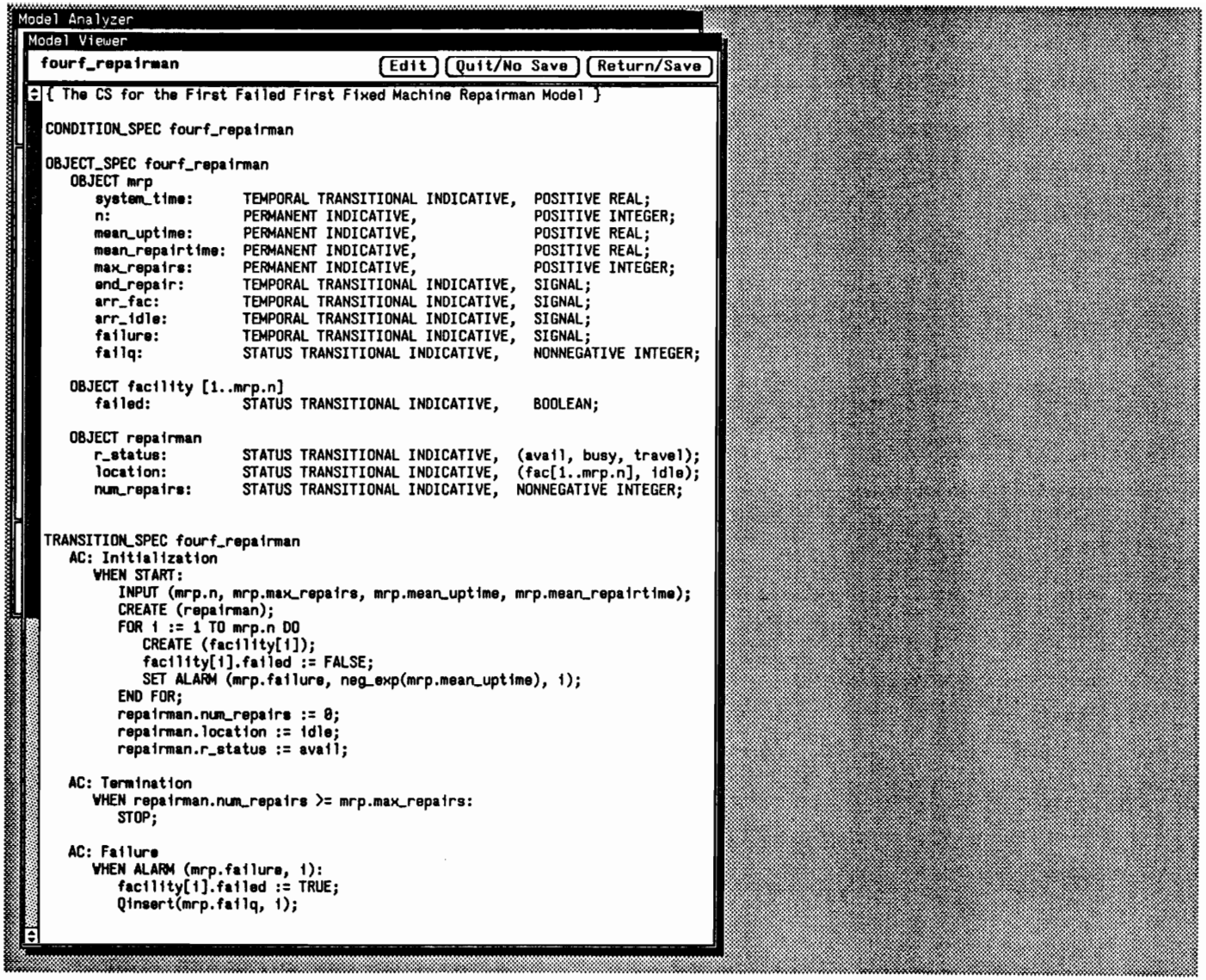


Figure 68. The Top Section of the First Failed, First Fixed Machine Repairman Condition Specification in the Model Viewer

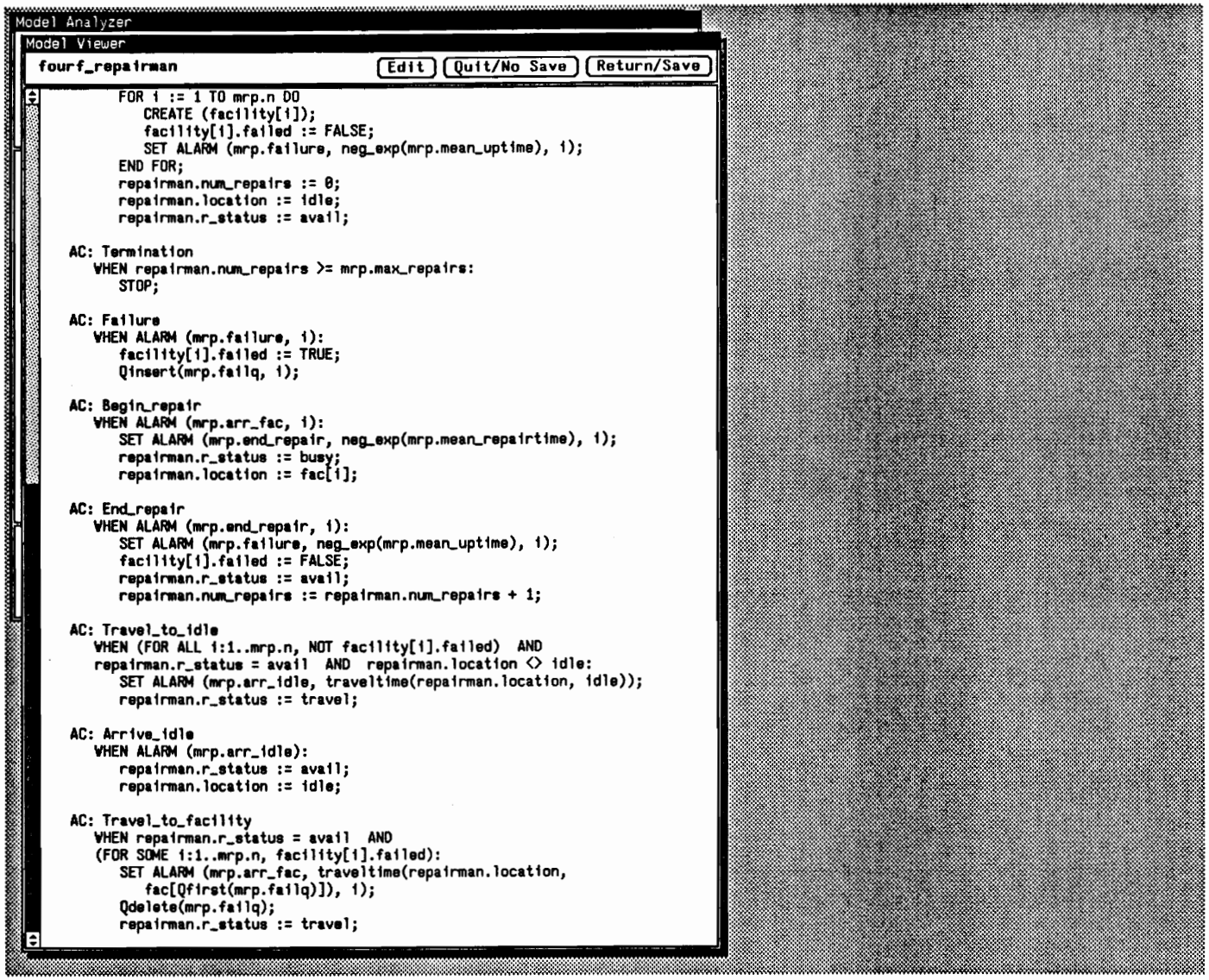


Figure 69. The Bottom Section of the First Failed, First Fixed Machine Repairman Condition Specification in the Model Viewer

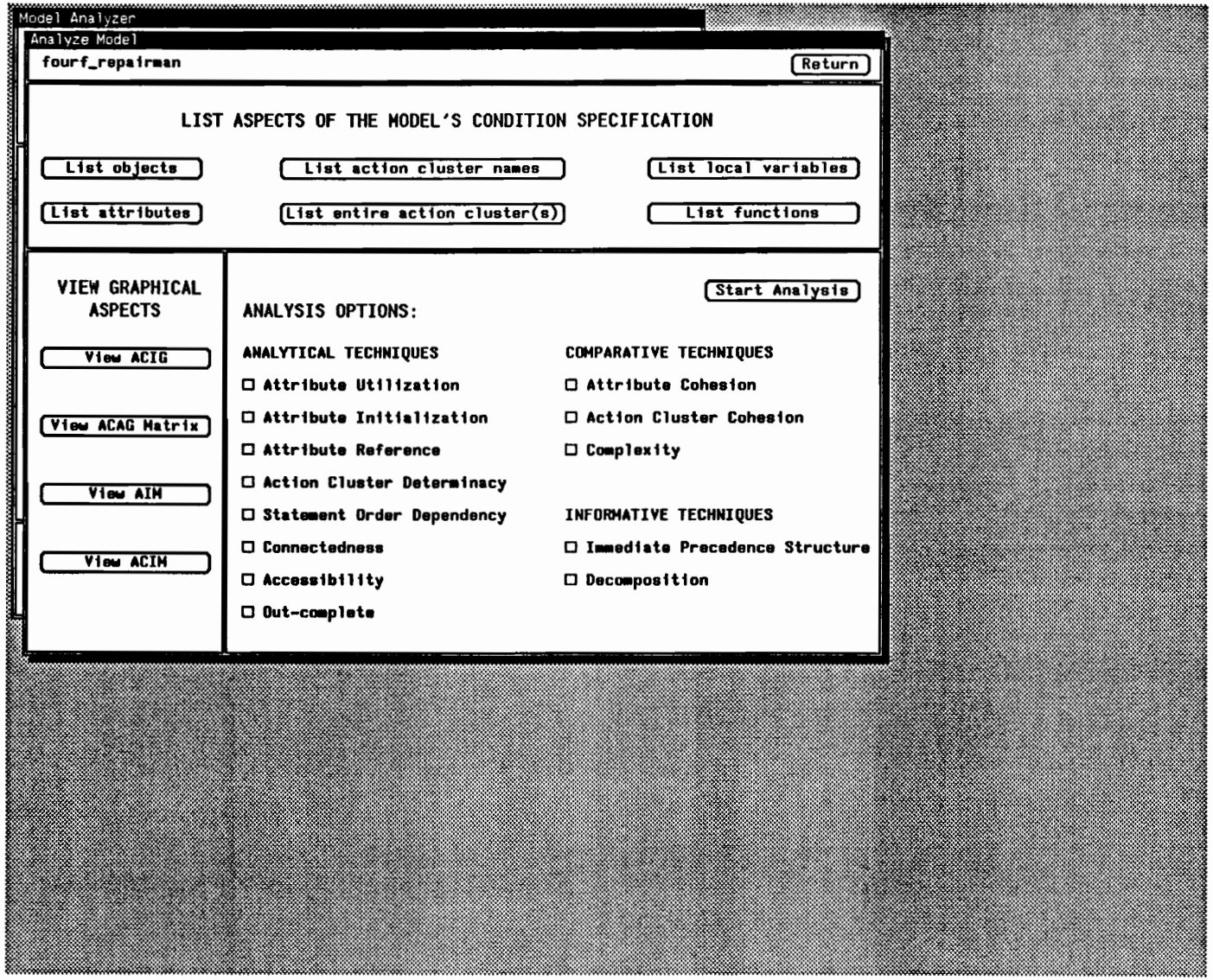


Figure 70. The Analyze Model Panel of the Model Analyzer

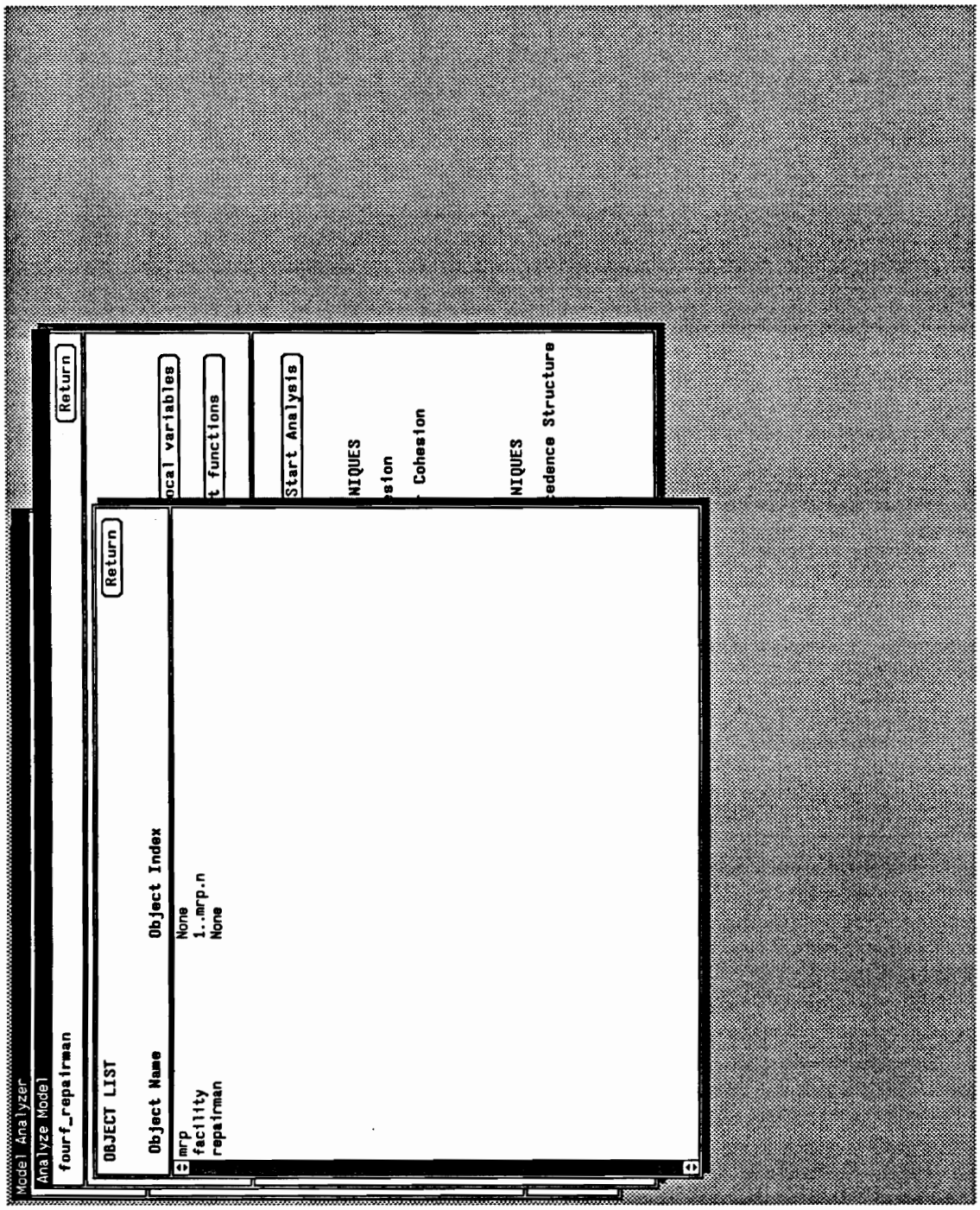


Figure 71. First Failed, First Fixed Machine Repairman Object List

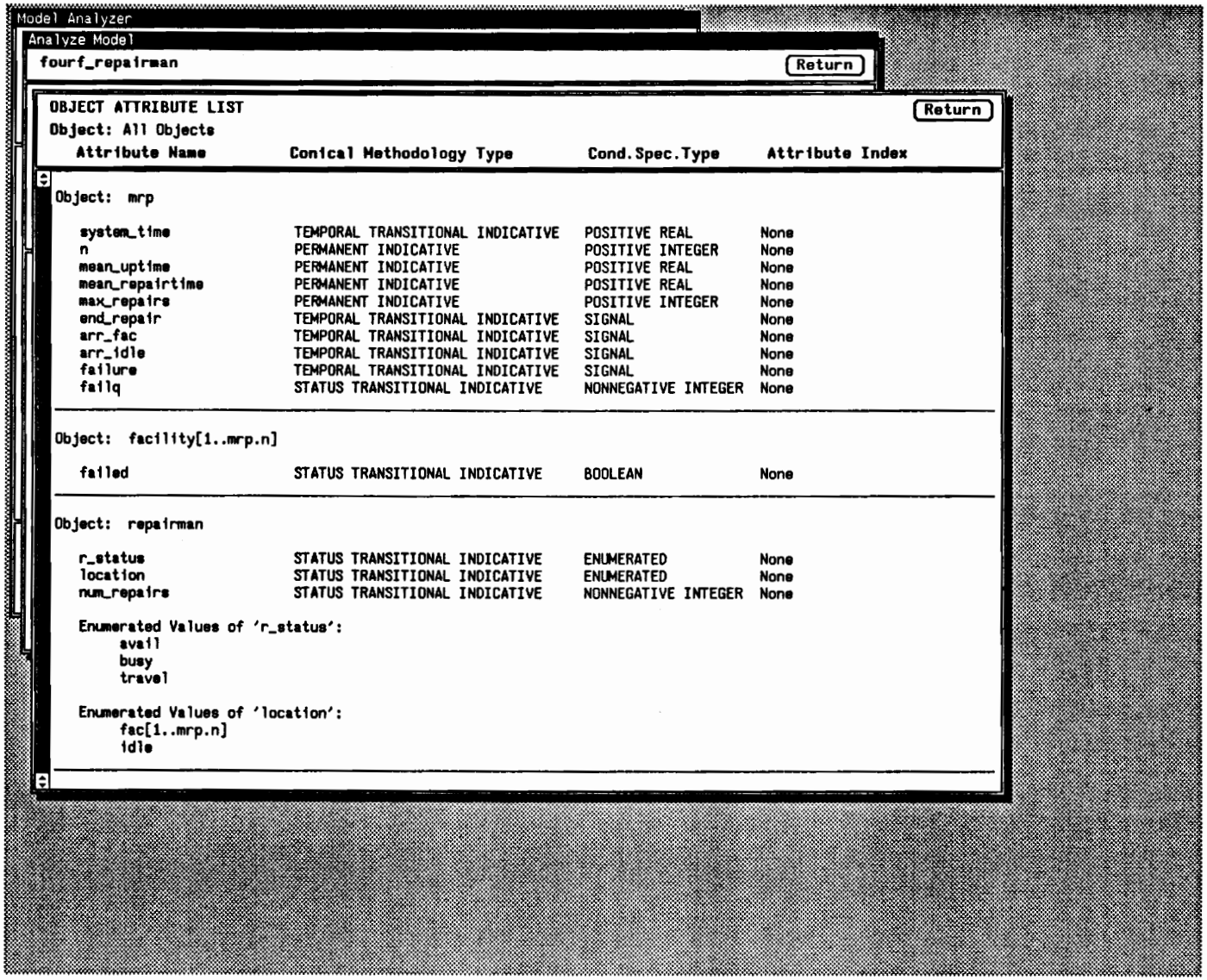


Figure 72. All Attributes of All Objects in First Failed, First Fixed Machine Repairman

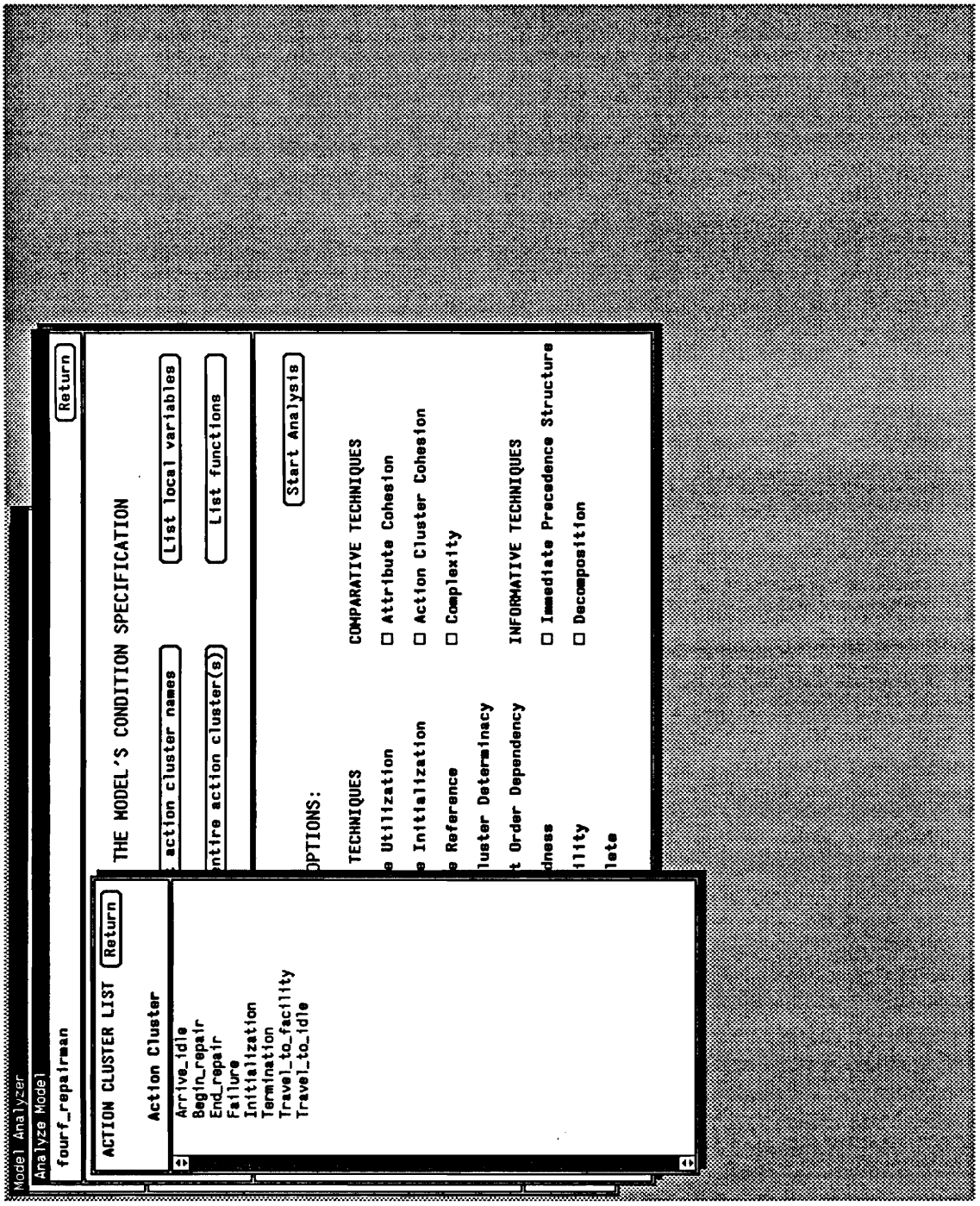


Figure 73. List of Action Cluster Names in First Failed, First Fixed Machine Repairman

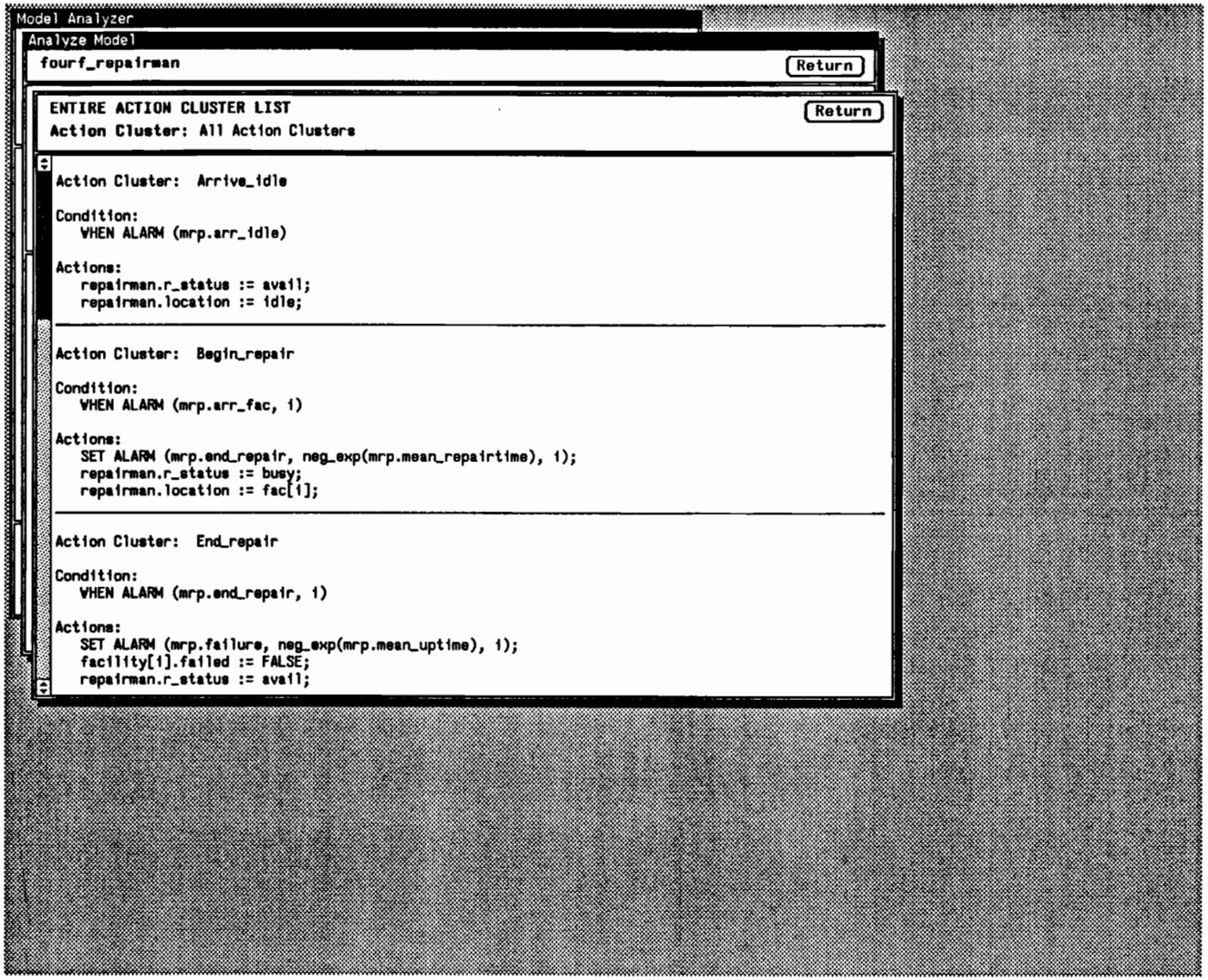


Figure 74. Conditions and Actions of All Action Clusters in First Failed, First Fixed Machine Repairman

attribute (as input, output, or control) within each action cluster is shown by selecting the "List attributes" button and the proper sequence of pull-right menu items: "Within Action Cluster(s)", "All Types", and "All Action Clusters". The modeler uses the scrollbar to see the entire list, shown in Figure 75.

To display the local variables used in the CS, the modeler selects the "List local variables" button and the "All Action Clusters" menu item. The local variable *i* is used in six of the eight action clusters (Figure 76). The modeler determines the functions used in the specification by selecting the "List functions" button and the "All Action Clusters" menu item. Figure 77 shows the list of functions used in each AC of the First Failed, First Fixed Machine Repairman Model.

6.2 View Graphical Aspects of the Model

Although the modeler can detect errors in the specification by listing the components of the model, graphically viewing the relationships among the components may find errors not seen in the listings. To show all possible interactions among the action clusters in the model, the unsimplified ACIG is viewed by selecting the "View ACIG" button, the "Unsimplified" menu item, and any one of the three items of the "ACIG FORM:" menu. Figures 78, 79, and 80 show, respectively, the matrix form, circular form, and linear form of the First Failed, First Fixed Machine Repairman unsimplified ACIG.

The modeler finds it somewhat difficult to understand the logic of the model from this unsimplified ACIG, containing twenty-one edges. The ACIG must be simplified before any valuable information is seen. The modeler simplifies the ACIG by first selecting the "View ACIG" button, the "Unsimplified" menu item, and one of the three "ACIG FORM:" menu items. In this session, the modeler chose to view the linear form

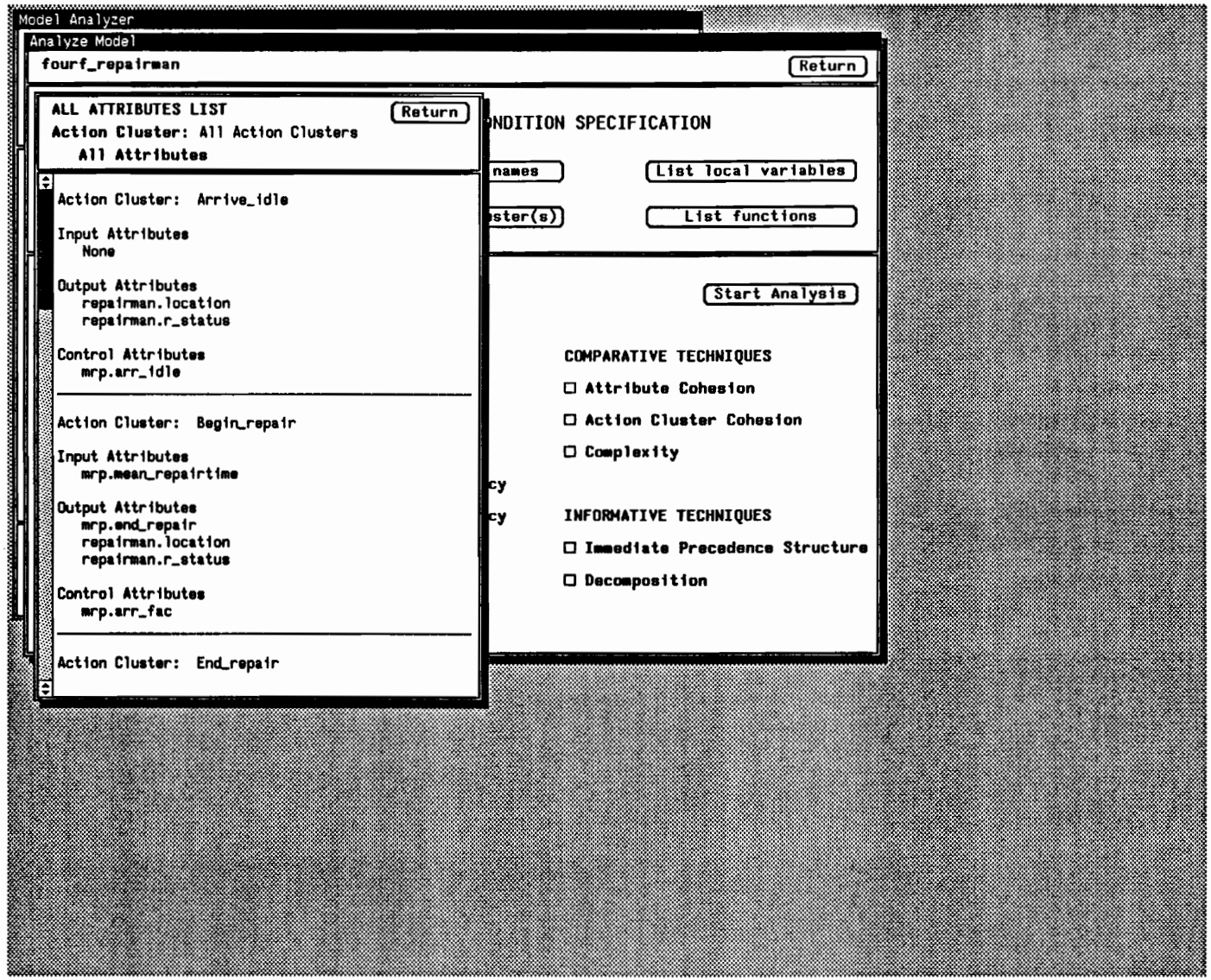


Figure 75. Classification of All Attributes of All Action Clusters in First Failed, First Fixed Machine Repairman

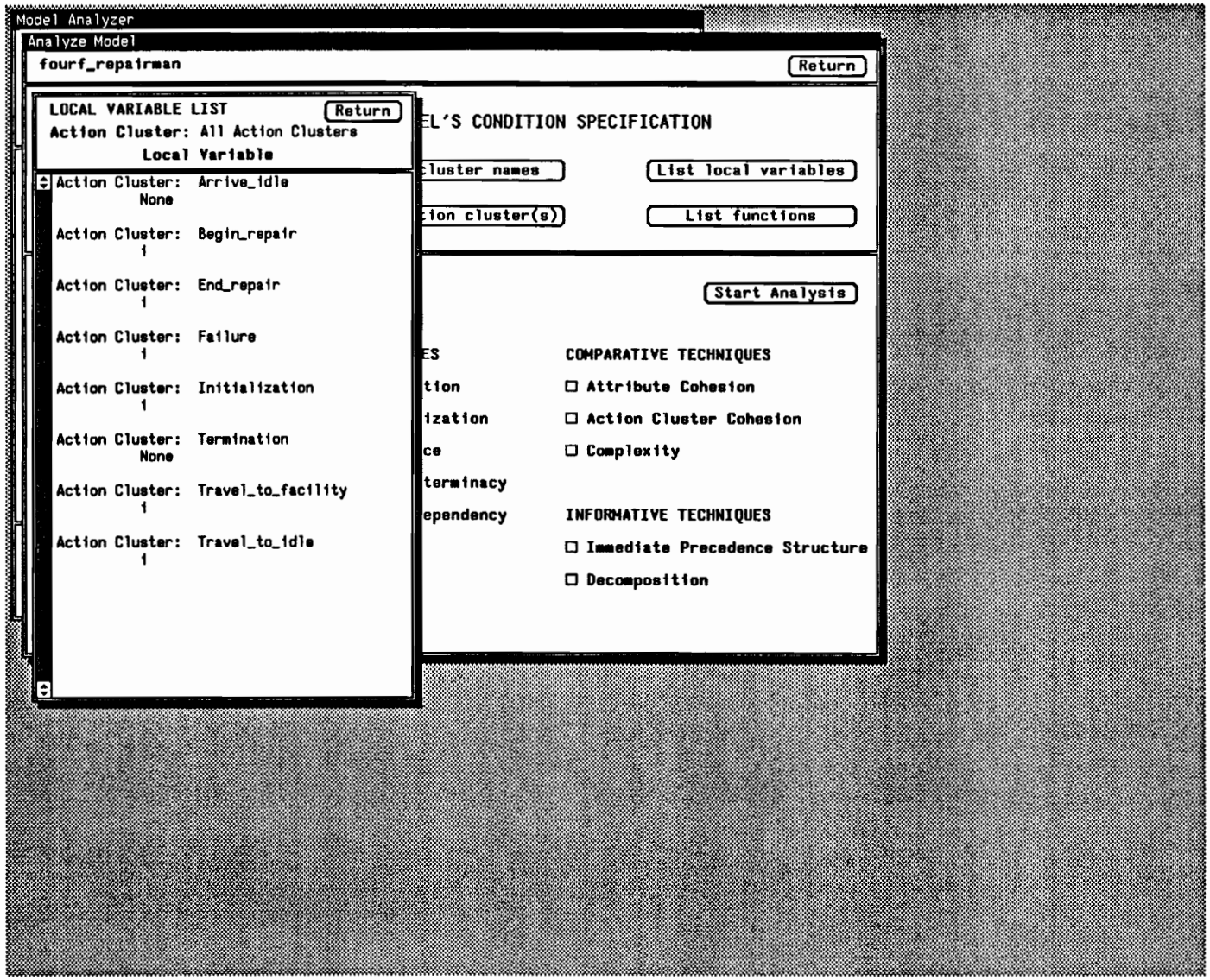


Figure 76. List of Local Variables in All Action Clusters of First Failed, First Fixed Machine Repairman

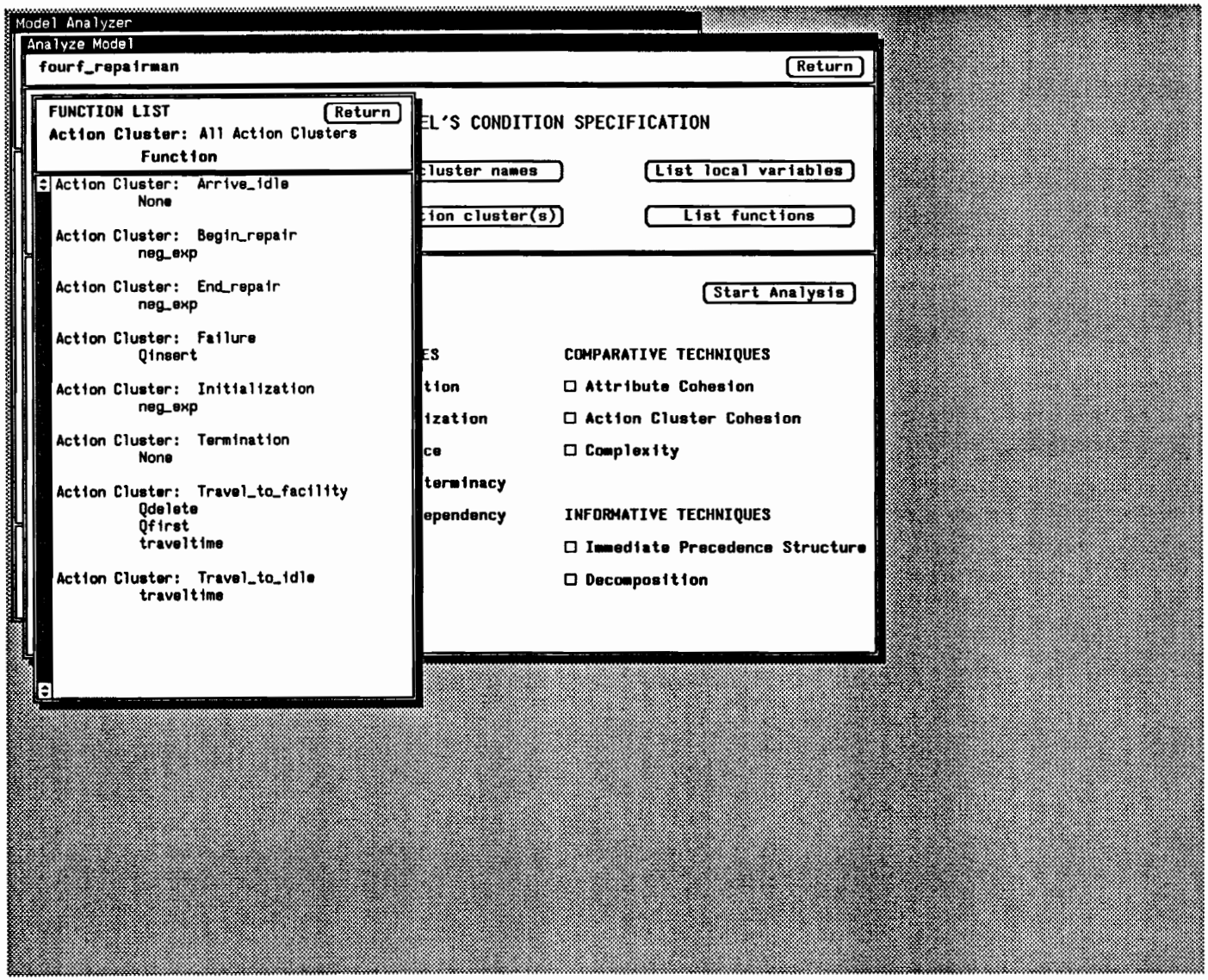


Figure 77. List of Functions in All Action Clusters of First Failed, First Fixed Machine Repairman

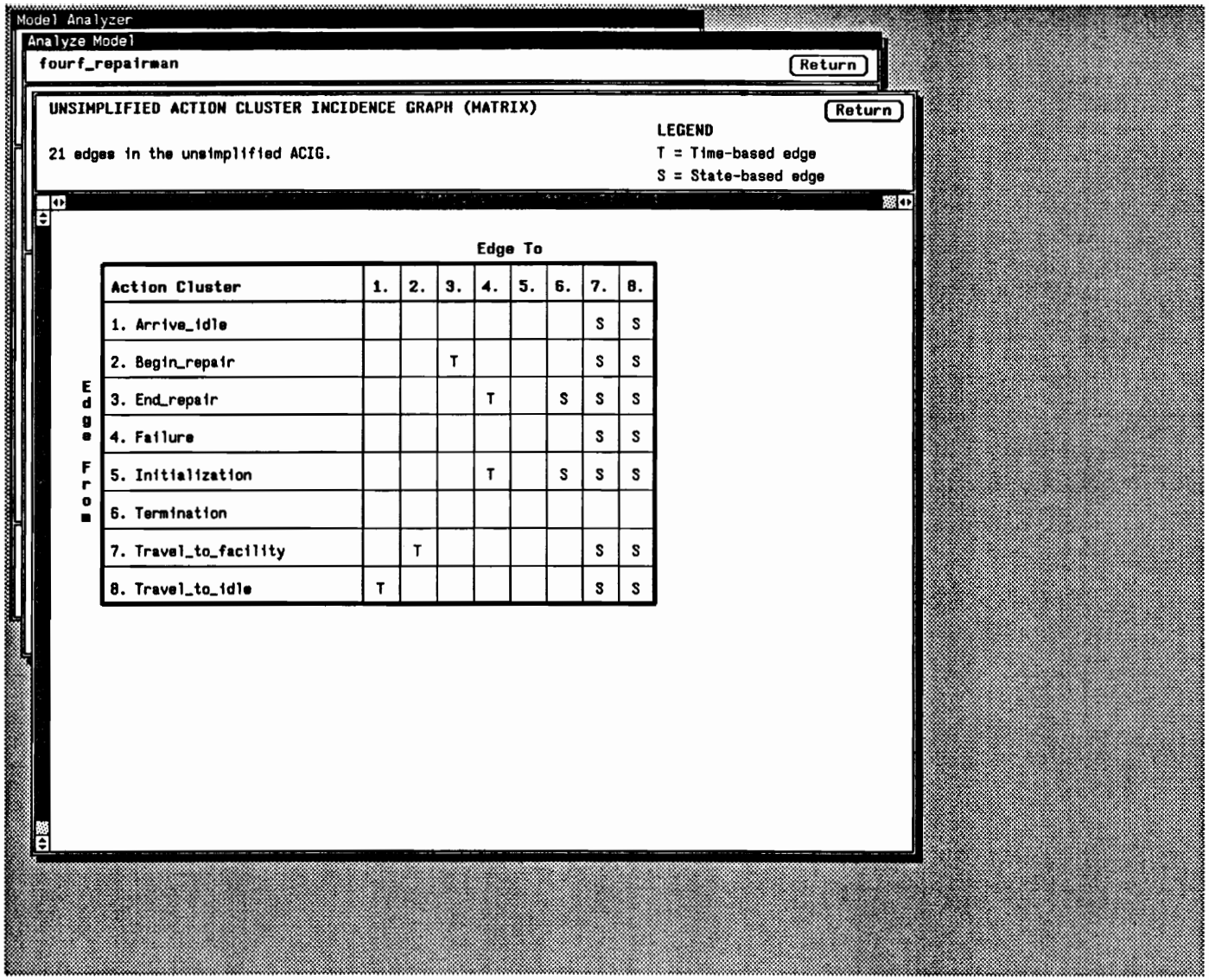


Figure 78. Matrix Form of First Failed, First Fixed Machine Repairman Unsimplified Action Cluster Incidence Graph

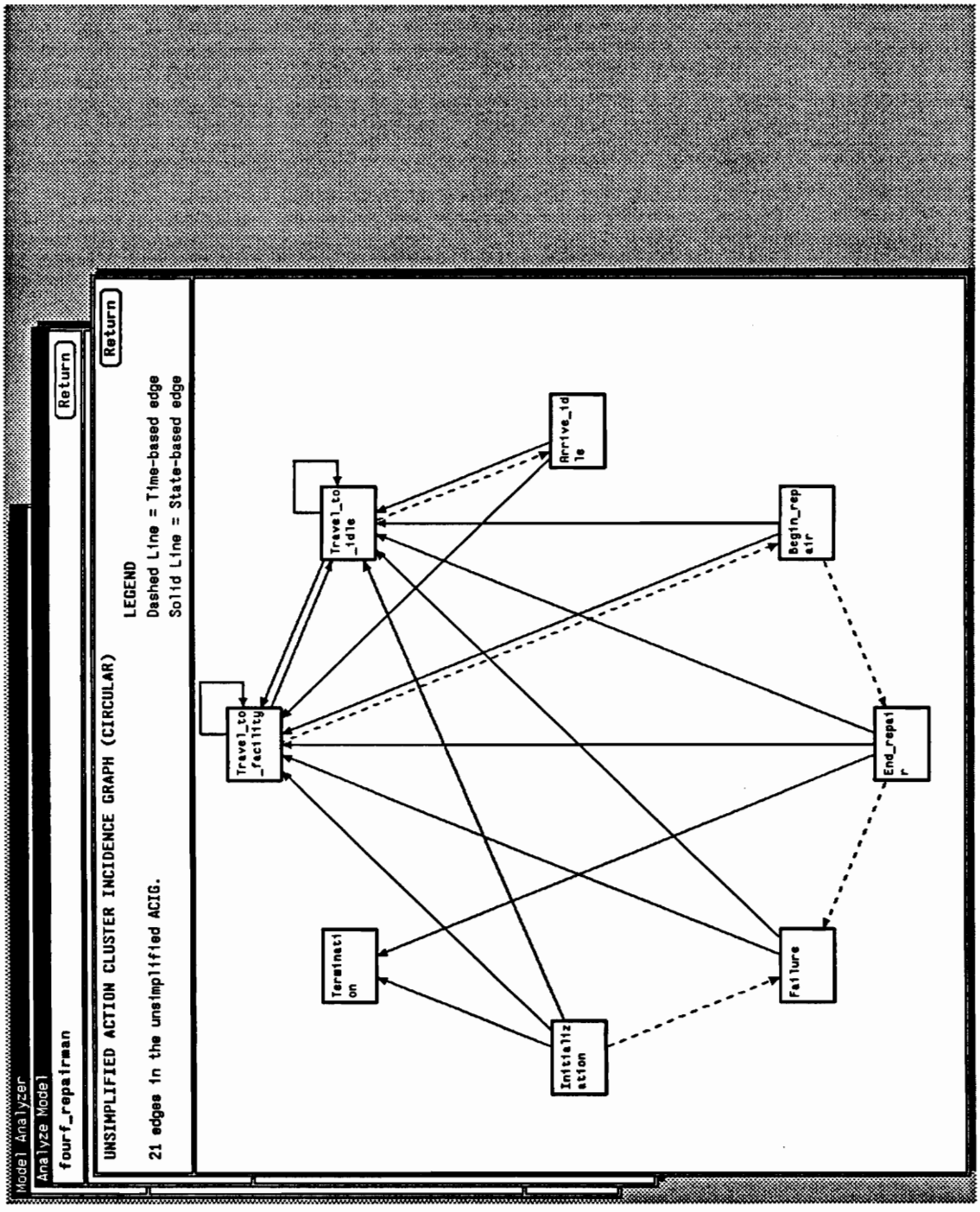


Figure 79. Circular Form of First Failed, First Fixed Machine Repairman Unsimplified Action Cluster Incidence Graph

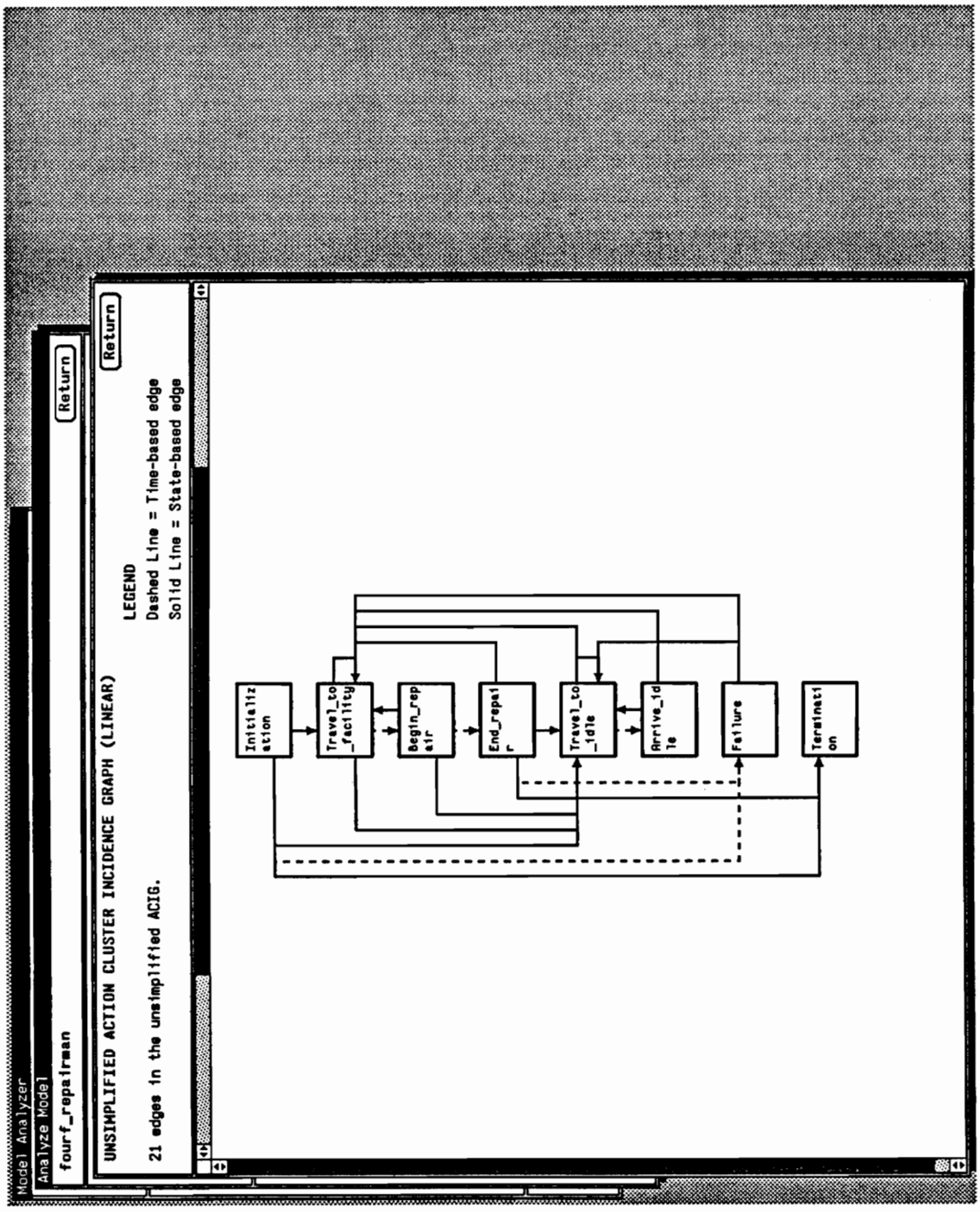


Figure 80. Linear Form of First Failed, First Fixed Machine Repairman Unsimplified Action Cluster Incidence Graph

of the unsimplified ACIG. After the simplified ACIG panel appears, the modeler selects the "Remove edge(s)" button, and the list of edges in the ACIG appears, as shown in Figure 81. At this point, the modeler determines which edges are infeasible by using the results, listed in Appendix F.2, of the expert system for the First Failed, First Fixed Machine Repairman Model. Each edge in the list in Figure 81, that is not listed in Appendix F.2, is selected and removed one at a time. After all the infeasible edges are removed from the list, the "Redraw graph" is selected and the ACIG is redrawn without the removed edges, shown in Figure 82. The simplified ACIG for the First Failed, First Fixed Machine Repairman Model contains ten edges, which is a 52% reduction of edges from the unsimplified ACIG. The modeler can easily trace the logic of the model in the linear form of the simplified ACIG in Figure 82. The simplified ACIG is also shown in its matrix and circular forms in Figures 83 and 84.

To view the relationship between attributes and action clusters graphically, the modeler selects the "View ACAG Matrix" button. The ACAG matrix, shown in Figure 85, displays how the attributes are used in each action cluster. This matrix contains the same information as listed in Figure 75, but in a more understandable form.

The modeler selects the "View AIM" button to view the interactions among attributes graphically. The AIM in Figure 86 shows that the attributes "mrp.n", "mrp.system_time", and "mrp.max_repairs" do not directly influence the value of any attribute, but "mrp.max_repairs" does influence the termination of the model. Also, the values of "mrp.failq" and "mrp.system_time" are not affected by any attributes. The values of "mrp.max_repairs", "mrp.mean_repairtime", "mrp.mean_uptime", and "mrp.n" can only be possibly affected by "mrp.mean_uptime", but further analysis of the CS shows that "mrp.mean_uptime" only directly affects the value of "mrp.failure".

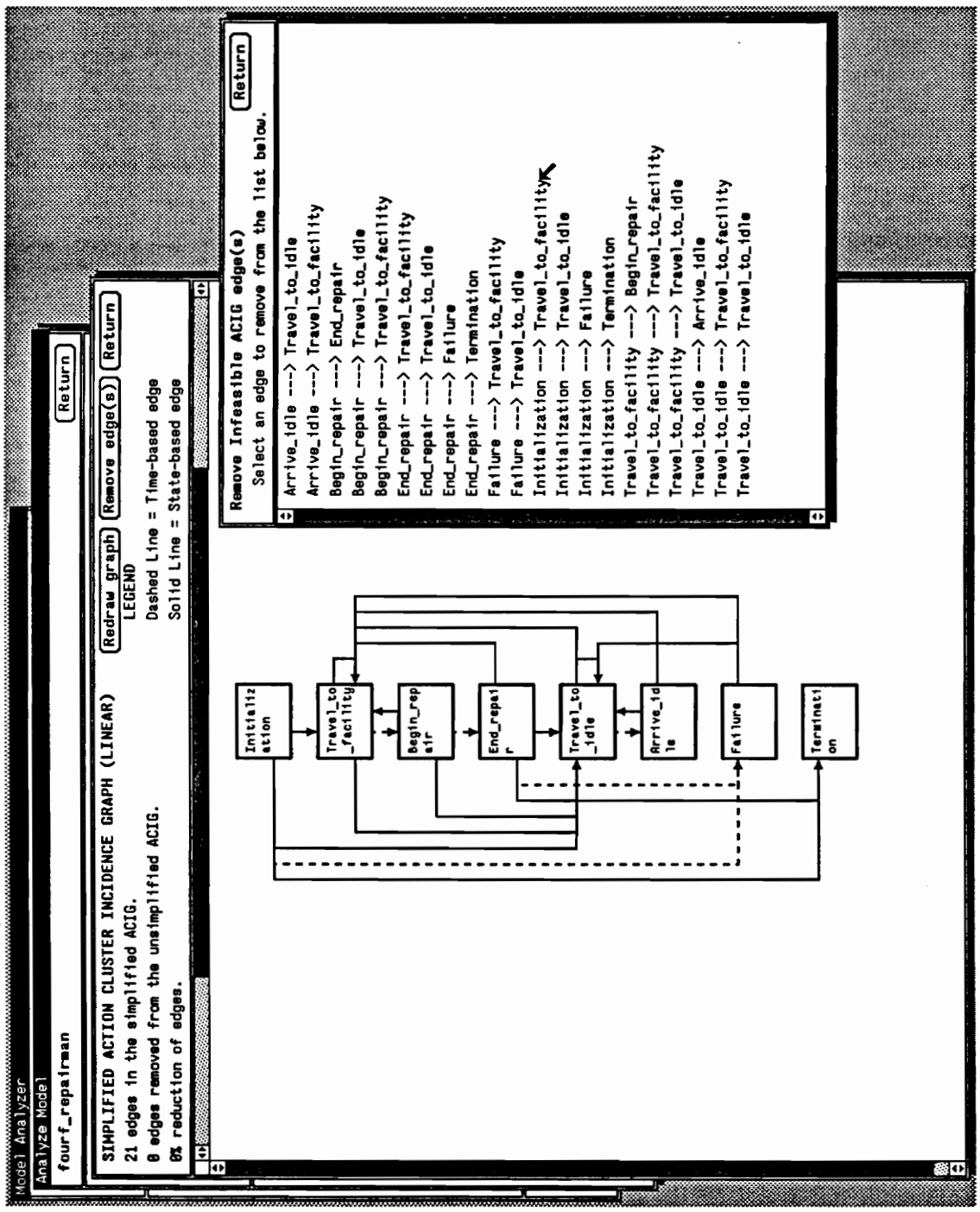


Figure 81. Edge List in First Failed, First Fixed Machine Repairman Simplified Action Cluster Incidence Graph Before Any Edges Are Removed

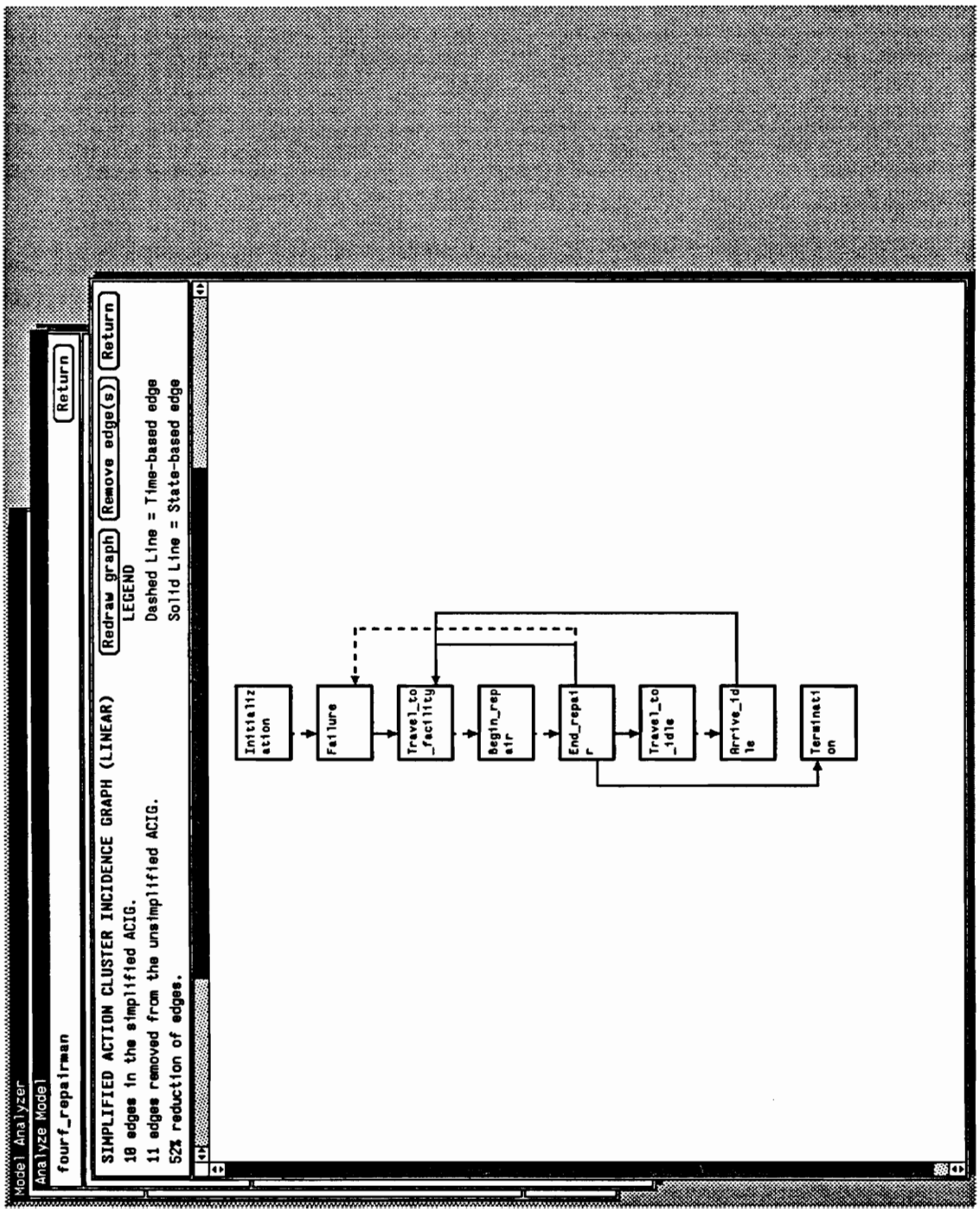


Figure 82. Linear Form of First Failed, First Fixed Machine Repairman Simplified Action Cluster Incidence Graph

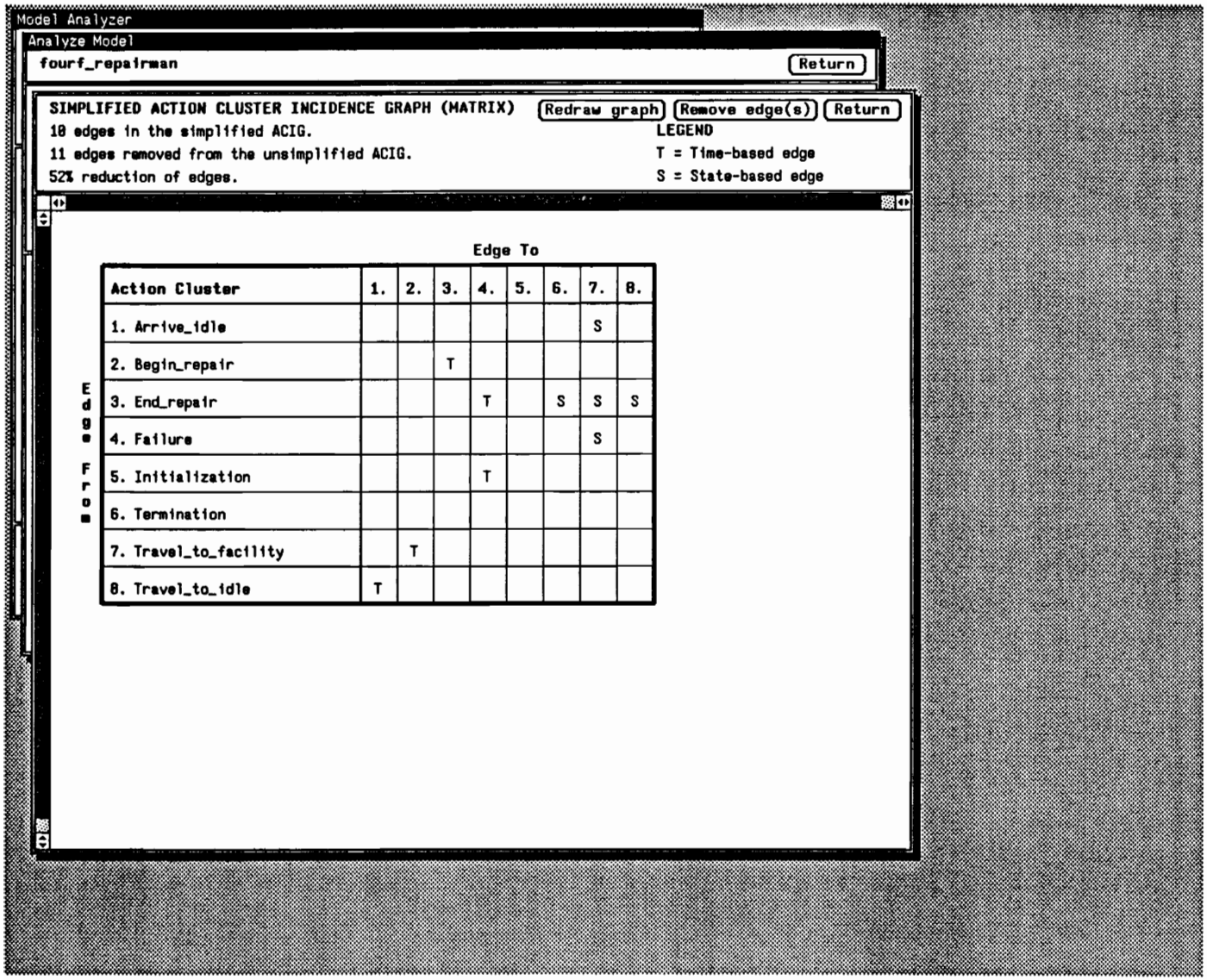


Figure 83. Matrix Form of First Failed, First Fixed Machine Repairman Simplified Action Cluster Incidence Graph

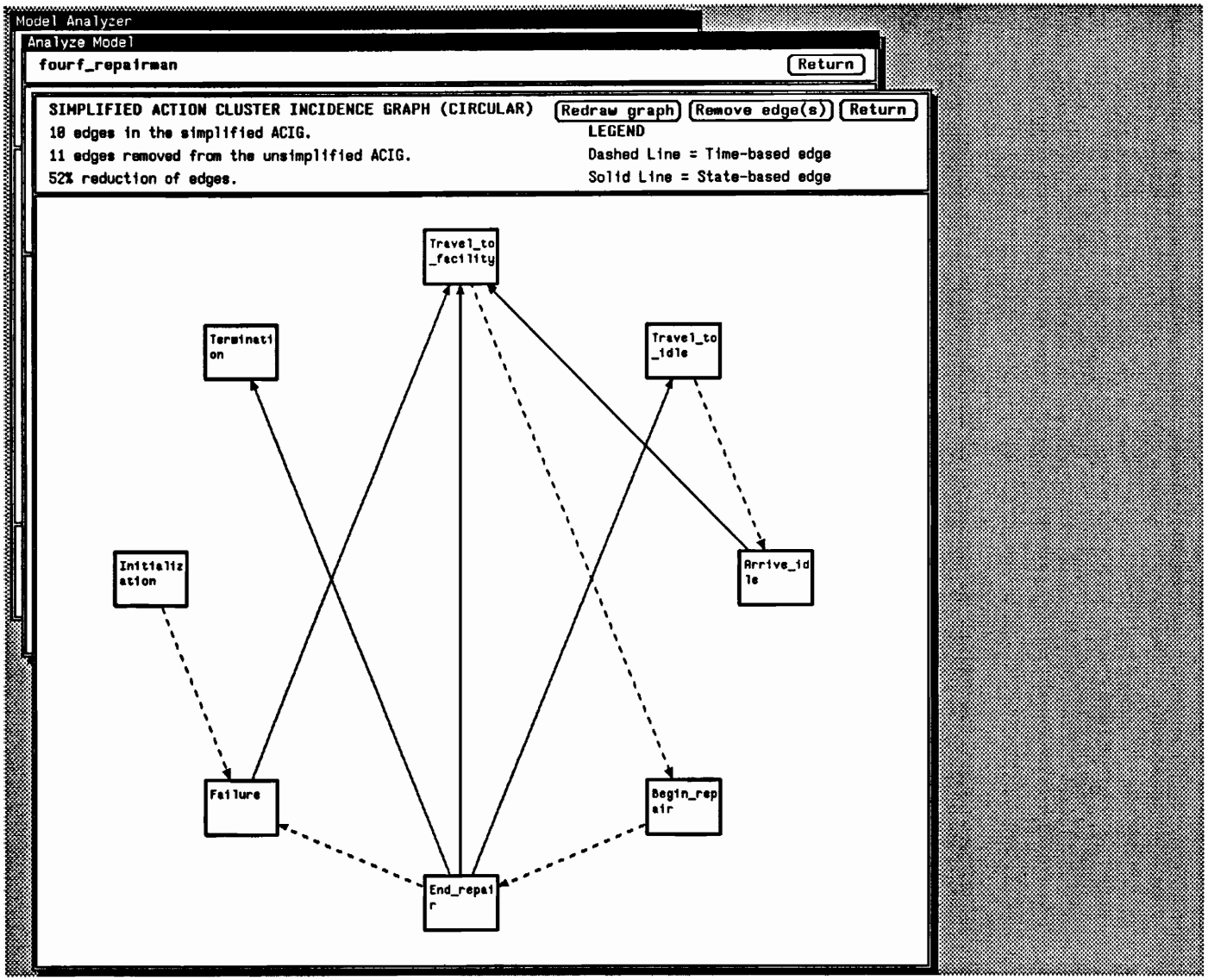
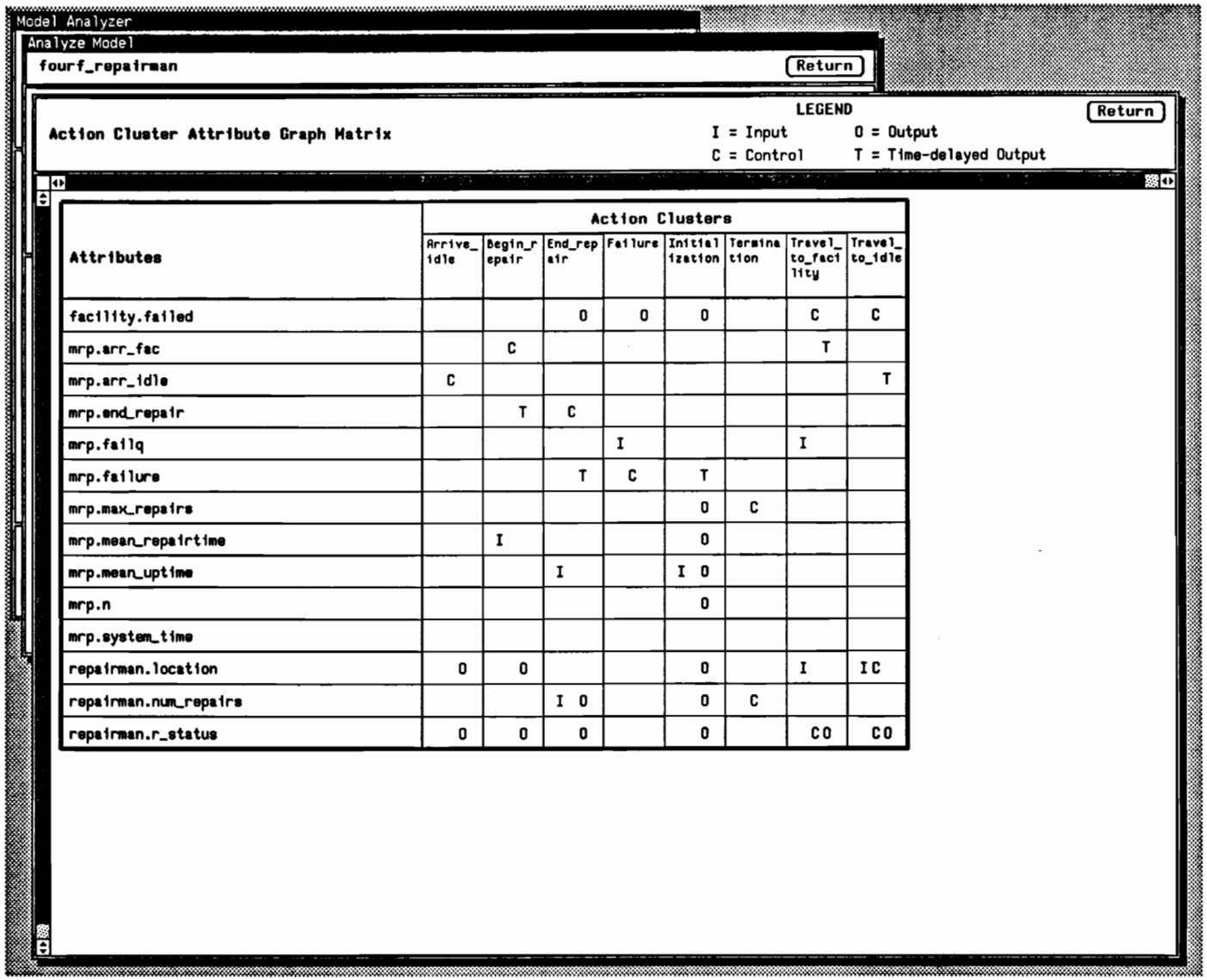
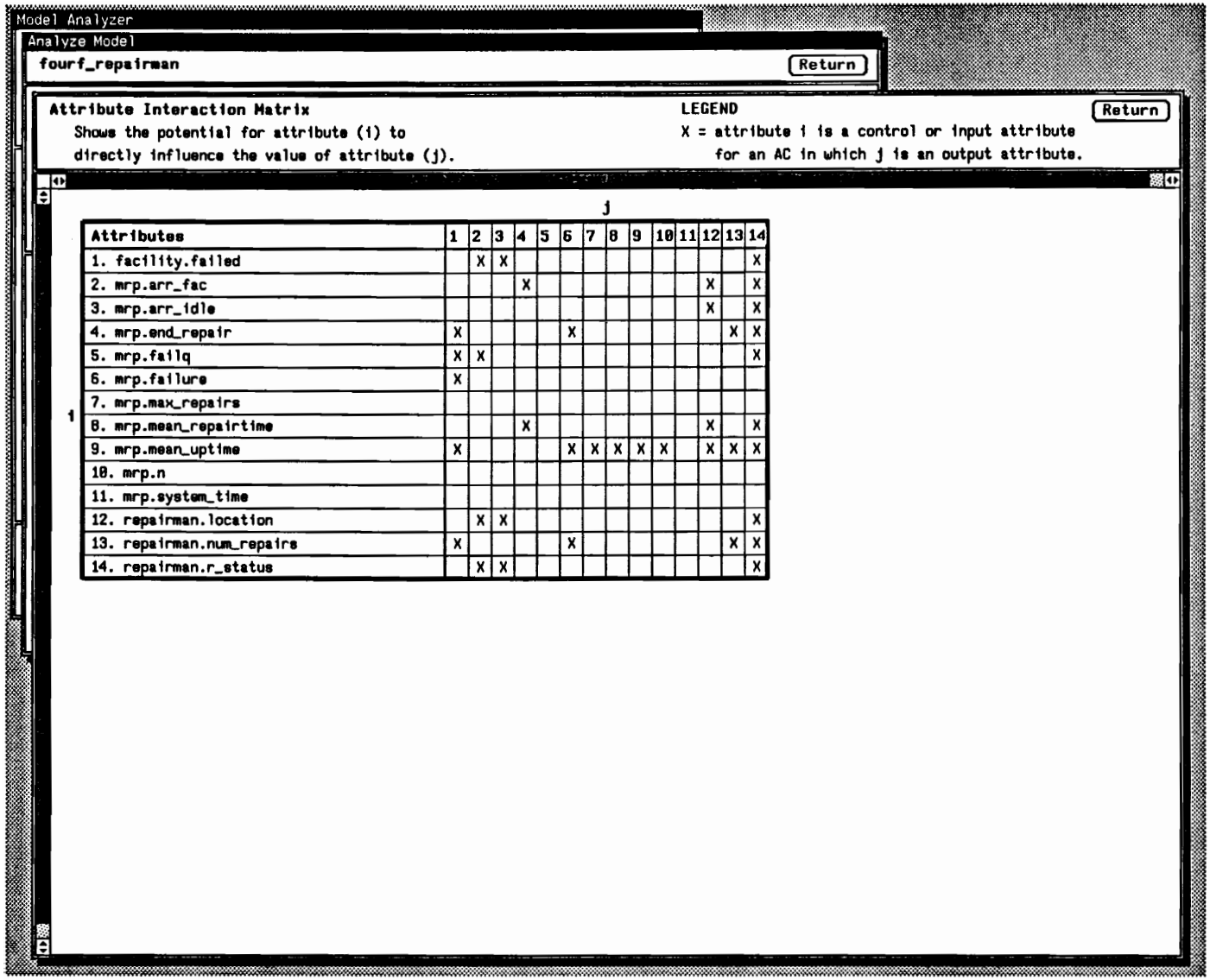


Figure 84. Circular Form of First Failed, First Fixed Machine Repairman Simplified Action Cluster Incidence Graph



Attributes	Action Clusters							
	Arrive_idle	Begin_repair	End_repair	Failure	Initialization	Termination	Travel_to_facility	Travel_to_idle
facility.failed			O	O	O		C	C
mrp.arr_fac		C					T	
mrp.arr_idle	C							T
mrp.end_repair		T	C					
mrp.failq				I			I	
mrp.failure			T	C	T			
mrp.max_repairs					O	C		
mrp.mean_repairtime		I			O			
mrp.mean_uptime			I		I O			
mrp.n					O			
mrp.system_time								
repairman.location	O	O			O		I	IC
repairman.num_repairs			I O		O	C		
repairman.r_status	O	O	O		O		CO	CO

Figure 85. Matrix Form of First Failed, First Fixed Machine Repairman Action Cluster Attribute Graph



Attributes	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1. facility.failed		X	X											X
2. mrp.arr_fac				X								X		X
3. mrp.arr_idle												X		X
4. mrp.end_repair	X					X							X	X
5. mrp.failq	X	X												X
6. mrp.failure	X													
7. mrp.max_repairs														
8. mrp.mean_repairtime				X								X		X
9. mrp.mean_uptime	X					X	X	X	X	X		X	X	X
10. mrp.n														
11. mrp.system_time														
12. repairman.location		X	X											X
13. repairman.num_repairs	X					X							X	X
14. repairman.r_status		X	X											X

Figure 86. First Failed, First Fixed Machine Repairman Attribute Interaction Matrix

The modeler selects the "View ACIM" button to view which action clusters contain attributes that directly influence the values of attributes in other action clusters. Figure 87 shows that "Termination" AC cannot affect any AC, and the "Initialization" AC can only be affected by itself. Also, the "Arrive_idle" AC can only be affected by the "Travel_to_idle" AC.

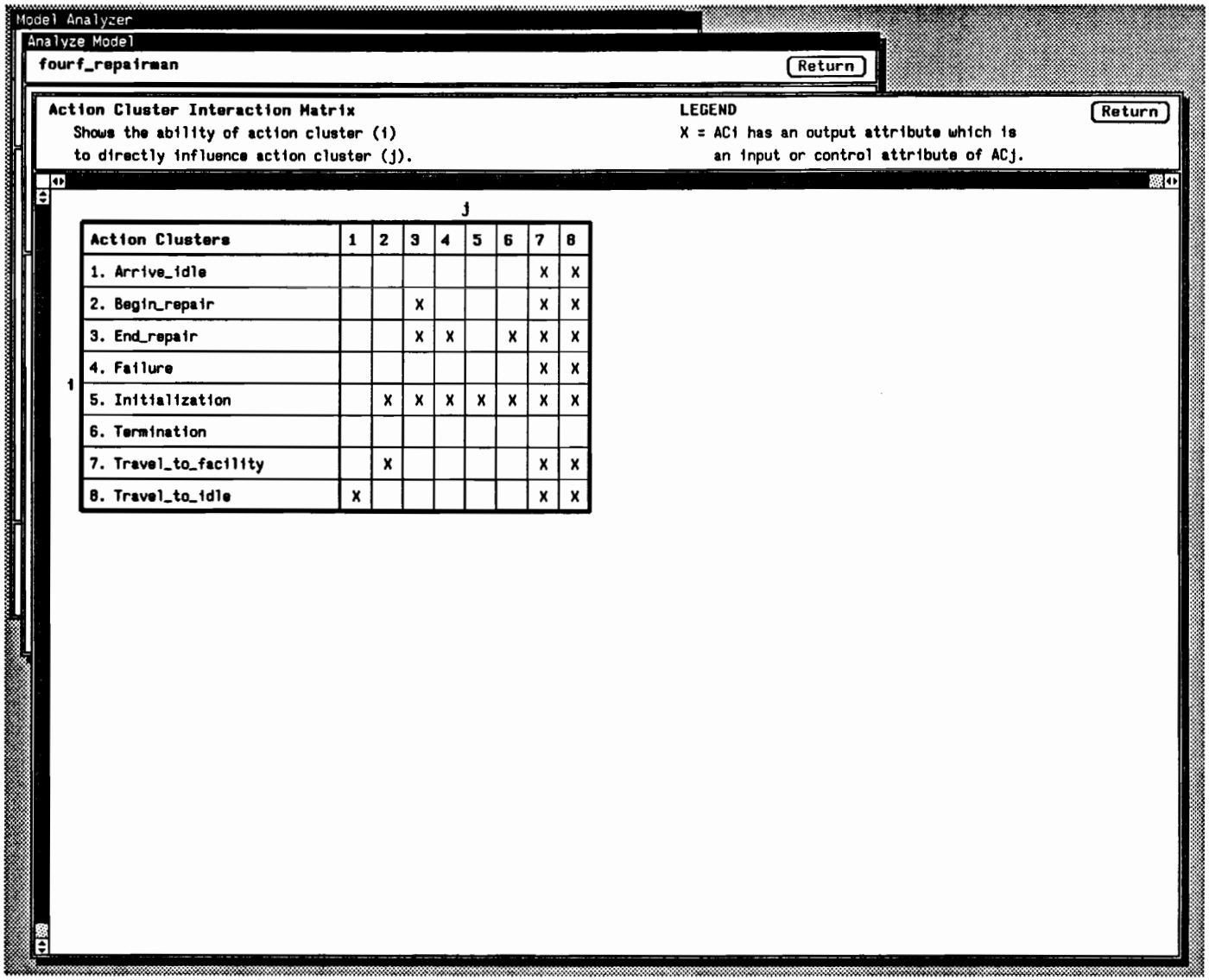
6.3 Perform Diagnostic Assistance

The last stage of the interactive analysis session is to perform diagnostic techniques upon the First Failed, First Fixed Machine Repairman Model. To obtain the most accurate results, many of these diagnostic techniques should only be performed after the ACIG is simplified. The analytical, comparative, and informative techniques are described in the following three sections.

6.3.1 Analytical Techniques

The modeler checks all eight of the analytical techniques listed in the Model Analyzer, shown in Figure 88, and then depresses the "Start Analysis" button. The results of all eight tests are displayed in one large scrollable text file, shown in Figure 89. The attribute utilization test, shown in Figure 89, states the attributes "mrp.system_time" and "mrp.n" are unutilized in the model. This does not signify an error in the specification since most model specifications do not specify when the "system time" needs to be updated because it is implied, and "mrp.n" is actually utilized as a range element of a *FOR* loop, which is one exception of this test.

Figure 90 shows that the attribute "mrp.failq" is determined uninitialized in the specification. The attribute "mrp.failq" represents the queue that holds the failed



Action Clusters	1	2	3	4	5	6	7	8
1. Arrive_idle							X	X
2. Begin_repair			X				X	X
3. End_repair			X	X		X	X	X
4. Failure							X	X
5. Initialization		X	X	X	X	X	X	X
6. Termination								
7. Travel_to_facility		X					X	X
8. Travel_to_idle	X						X	X

Figure 87. First Failed, First Fixed Machine Repairman Action Cluster Interaction Matrix

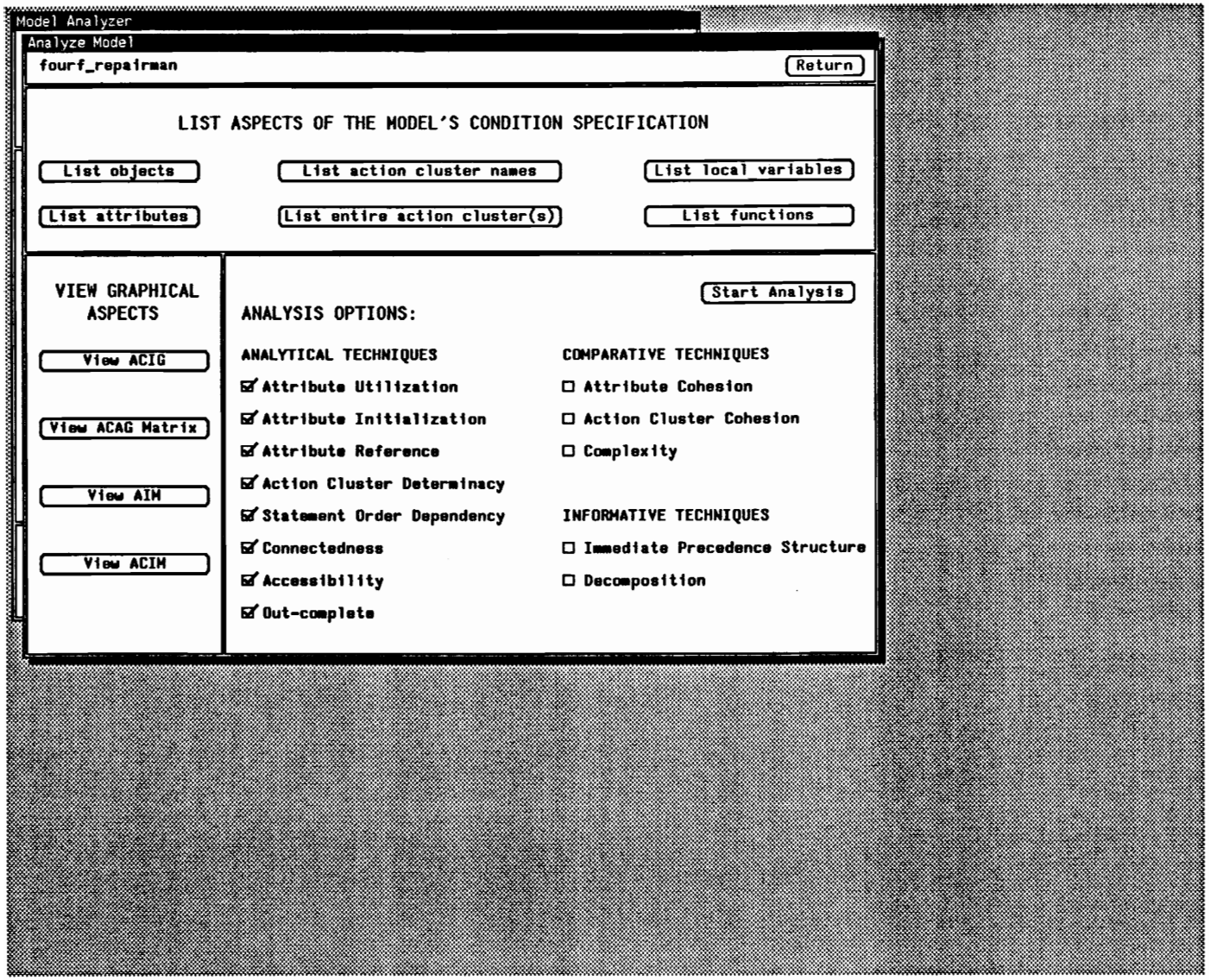


Figure 88. The Selected Analytical Techniques in the Model Analyzer

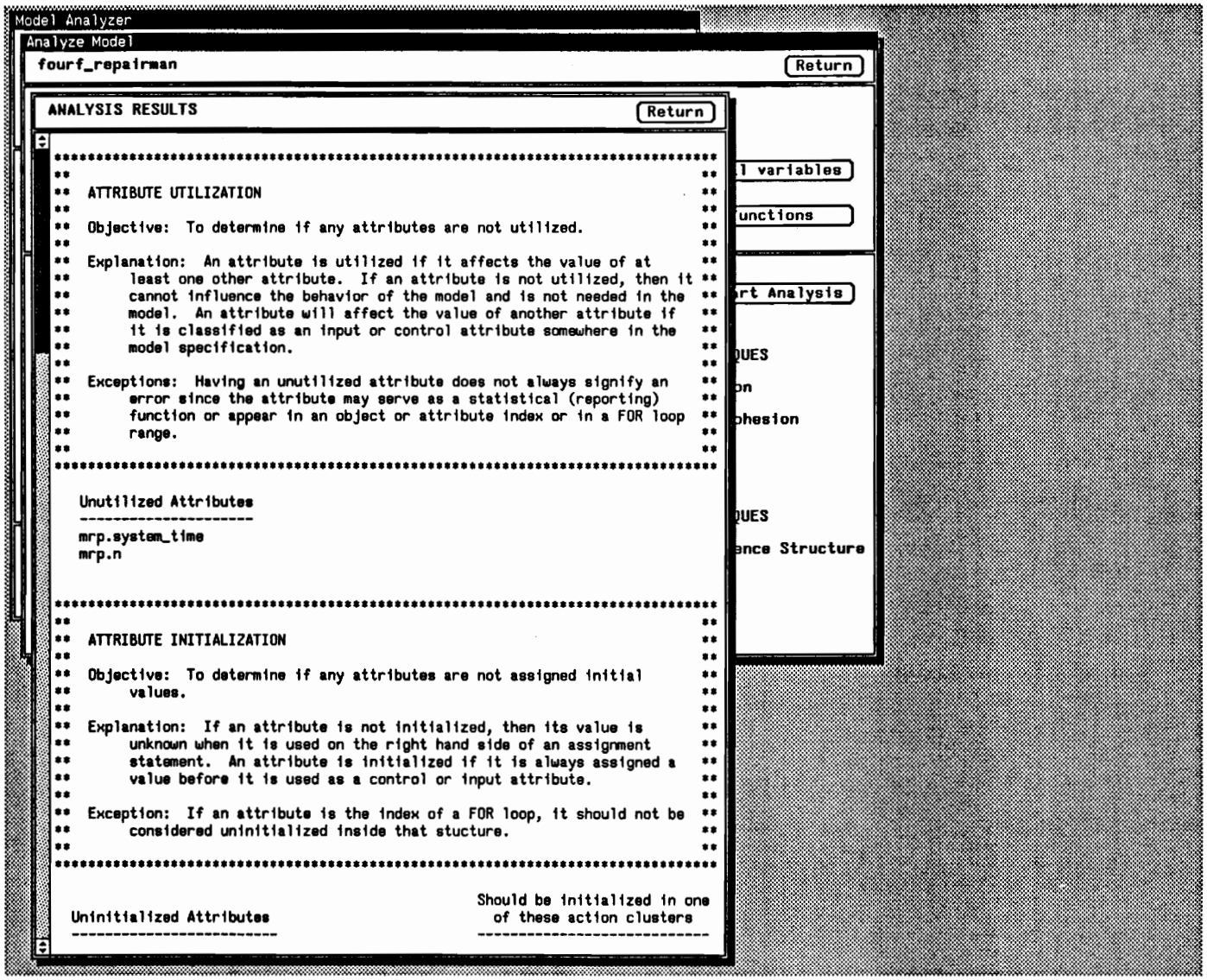


Figure 89. First Failed, First Fixed Machine Repairman Attribute Utilization Results

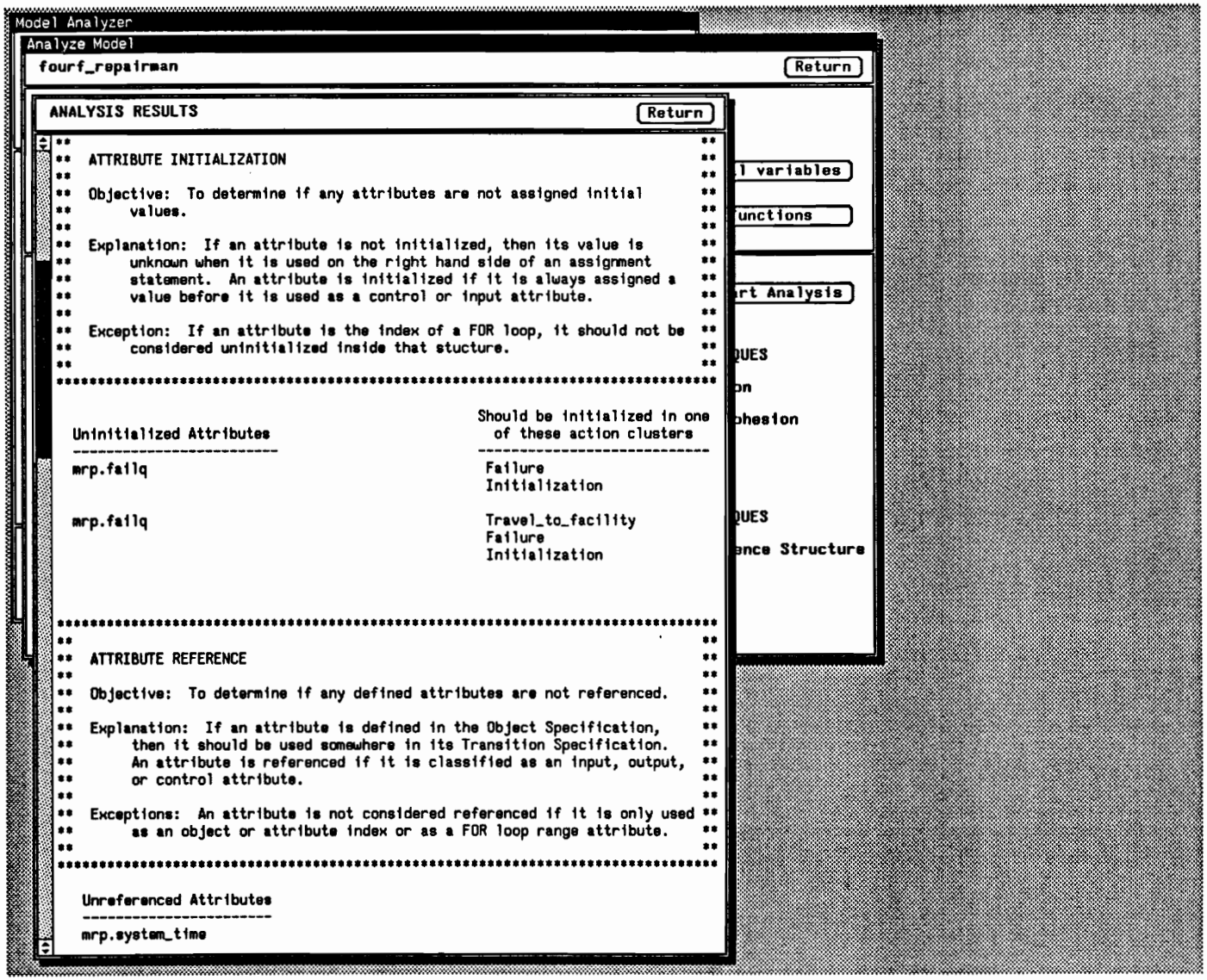


Figure 90. First Failed, First Fixed Machine Repairman Attribute Initialization and Attribute Reference Results

machines. Its structure is hidden from the specification, and therefore its initialization is also hidden from the specification.

The attribute "mrp.system_time" is not referenced (Figure 90) in the specification of the model, because the updating of the system time is assumed. The action cluster determinacy test, shown in Figure 91, states all action clusters contain the proper state changes to prevent the "infinite loop" syndrome.

The statement order dependency test, in Figure 91, determines the statement dependent action clusters are "End_repair" and "Initialization". The "End_repair" AC is actually not statement dependent because the attribute "repairman.num_repairs" is an input and output attribute in the same statement. The "Initialization" AC is statement dependent and must execute the *INPUT* statement before the *SET ALARM* statement to ensure proper execution of the model.

The First Failed, First Fixed Machine Repairman ACIG is fully connected and completely accessible from the "Initialization" AC, as shown in Figure 92. Figure 93 determines the CS as out-complete.

6.3.2 Comparative Techniques

After viewing the results of the analytical techniques, the modeler returns to the Analyze Model panel, unchecks the analytical techniques, checks the three comparative techniques, and selects the "Start Analysis" button. The results of the comparative techniques are also displayed in a scrollable panel, shown in Figure 94, which shows an attribute cohesion index of four and an action cluster cohesion index of four. Figure 95 shows a model complexity of 13.33.

The results do not tell the modeler anything valuable until these results are compared to the results of other models. Table 15 shows the results of the three

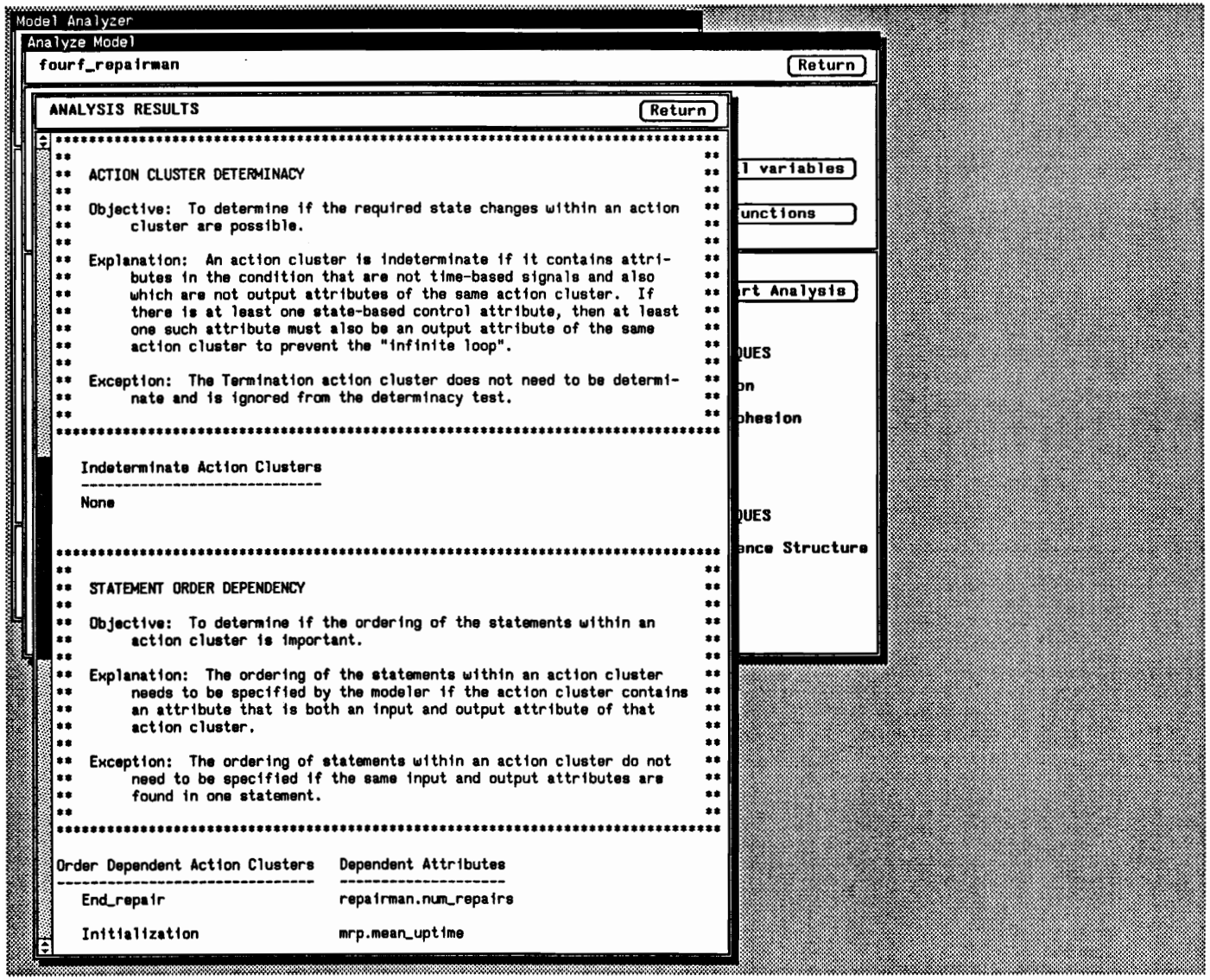


Figure 91. First Failed, First Fixed Machine Repairman Action Cluster Determinacy and Statement Order Dependency Results

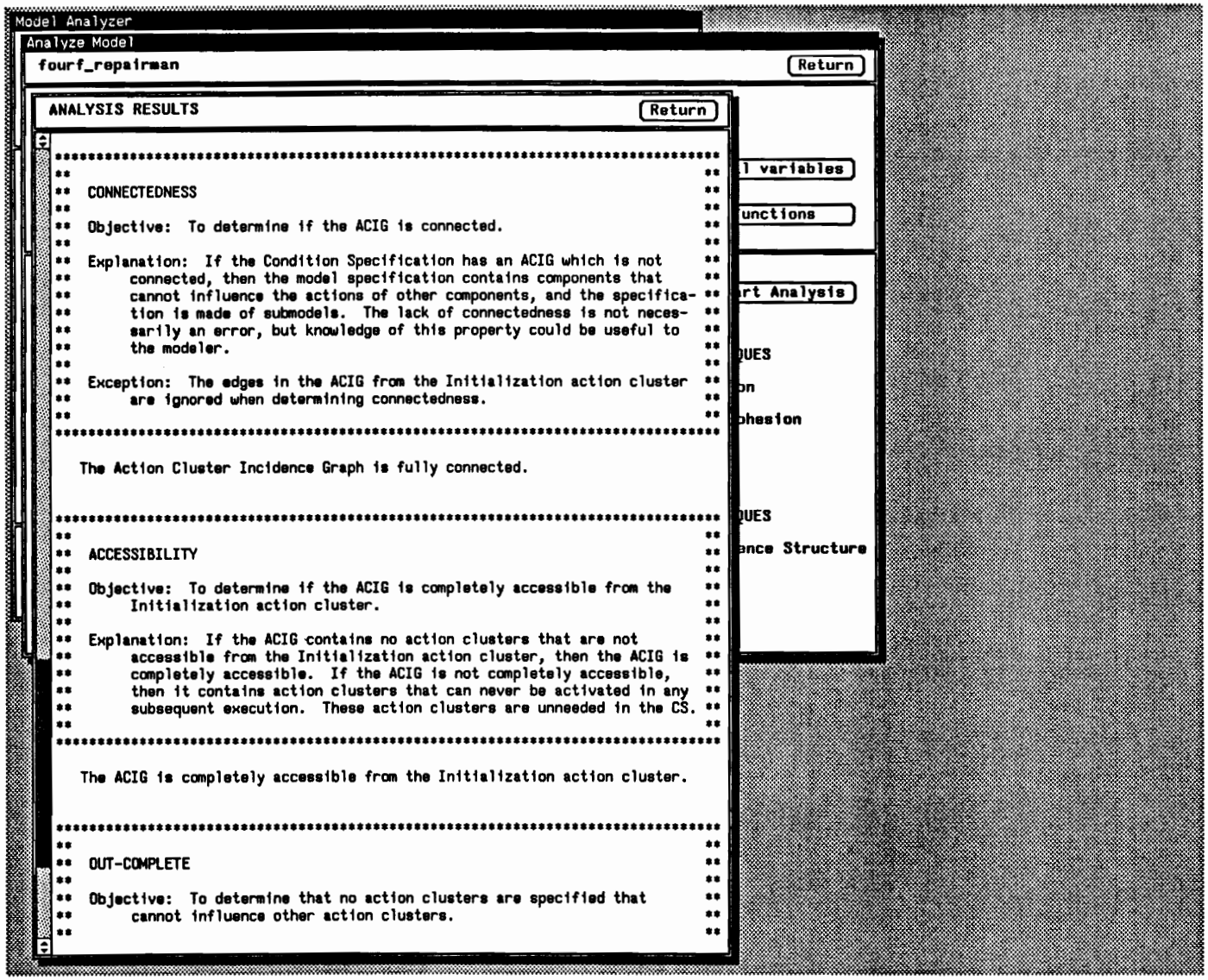


Figure 92. First Failed, First Fixed Machine Repairman Connectedness and Accessibility Results

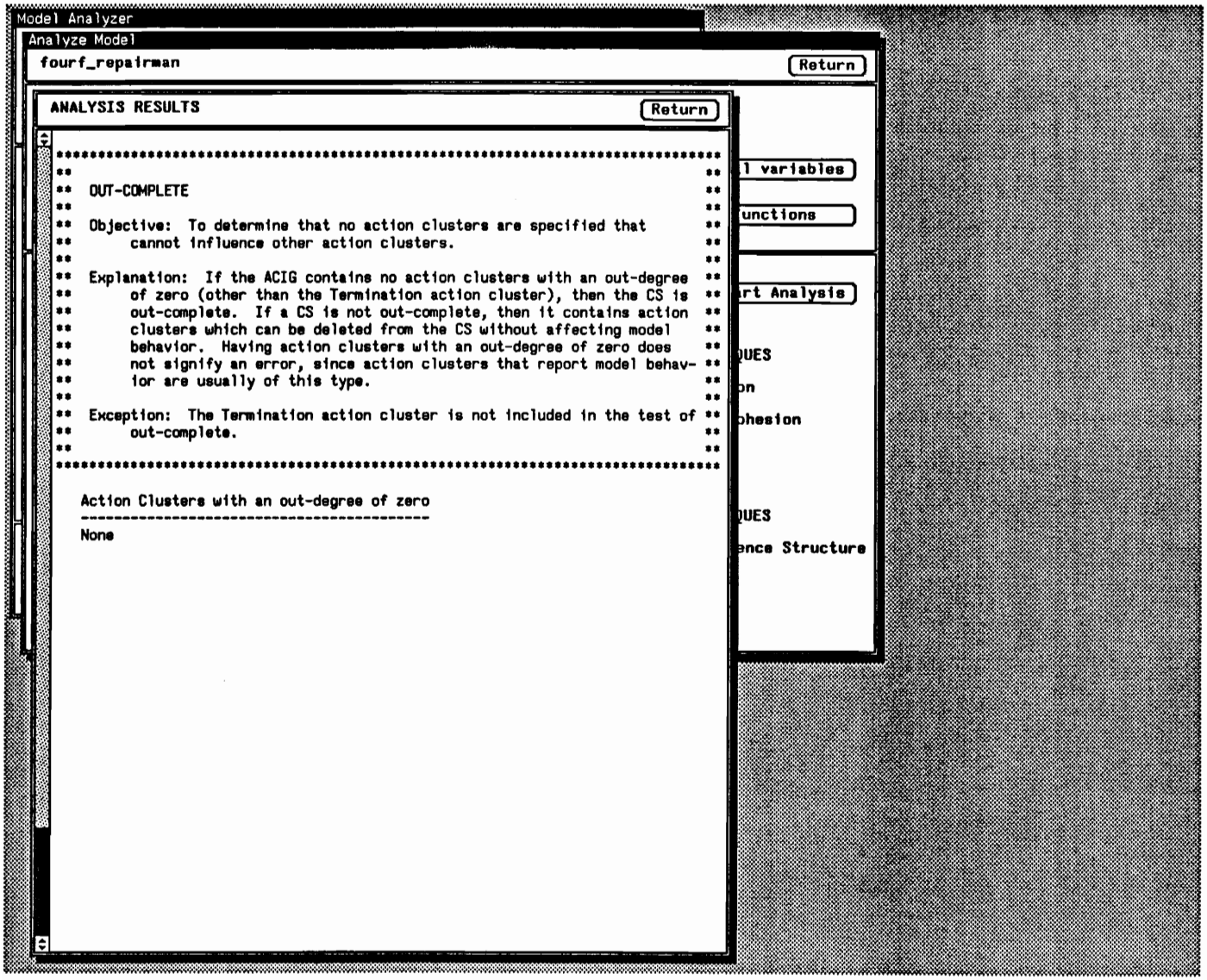


Figure 93. First Failed, First Fixed Machine Repairman Out-Completeness Results

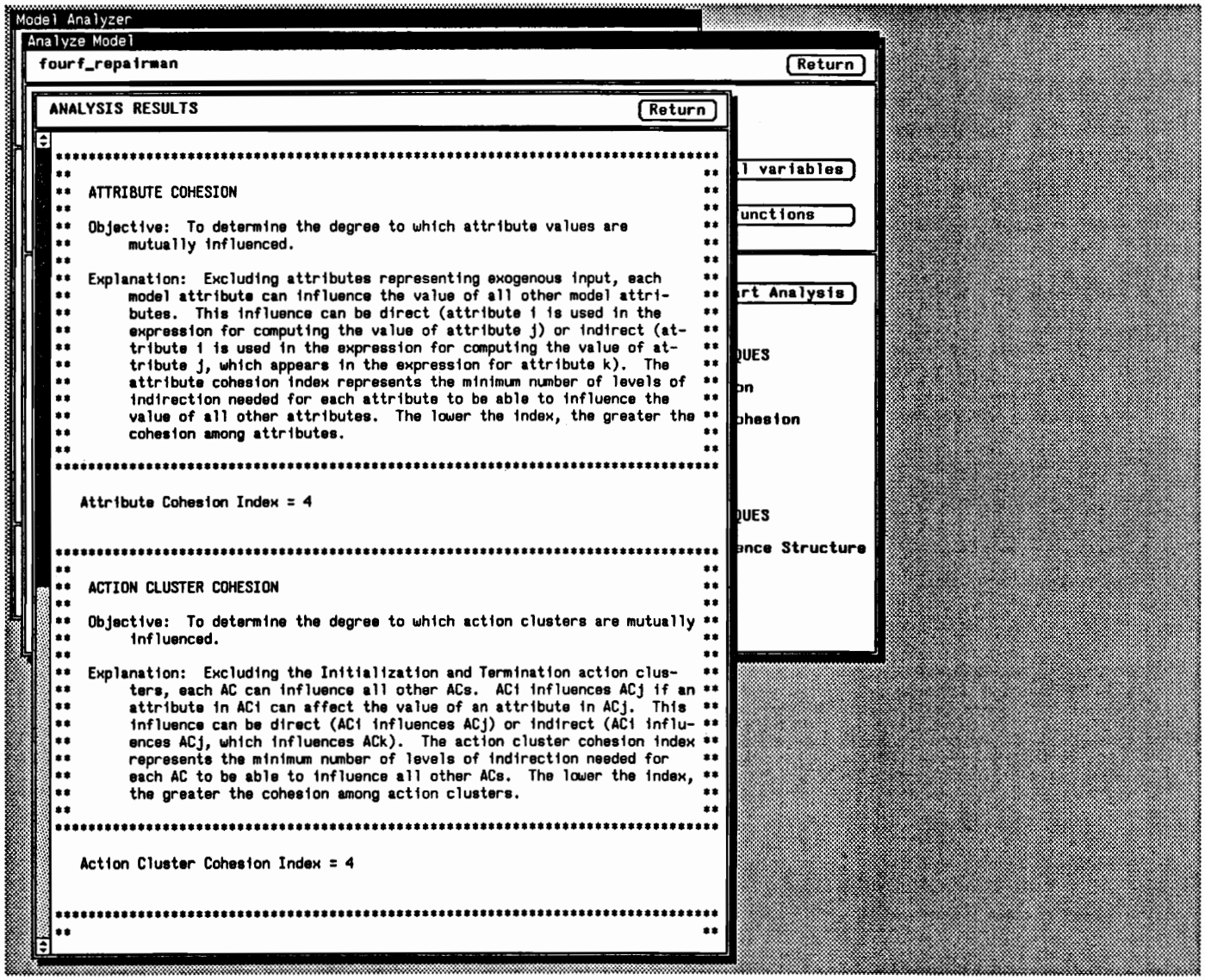


Figure 94. First Failed, First Fixed Machine Repairman Attribute Cohesion and Action Cluster Cohesion Results

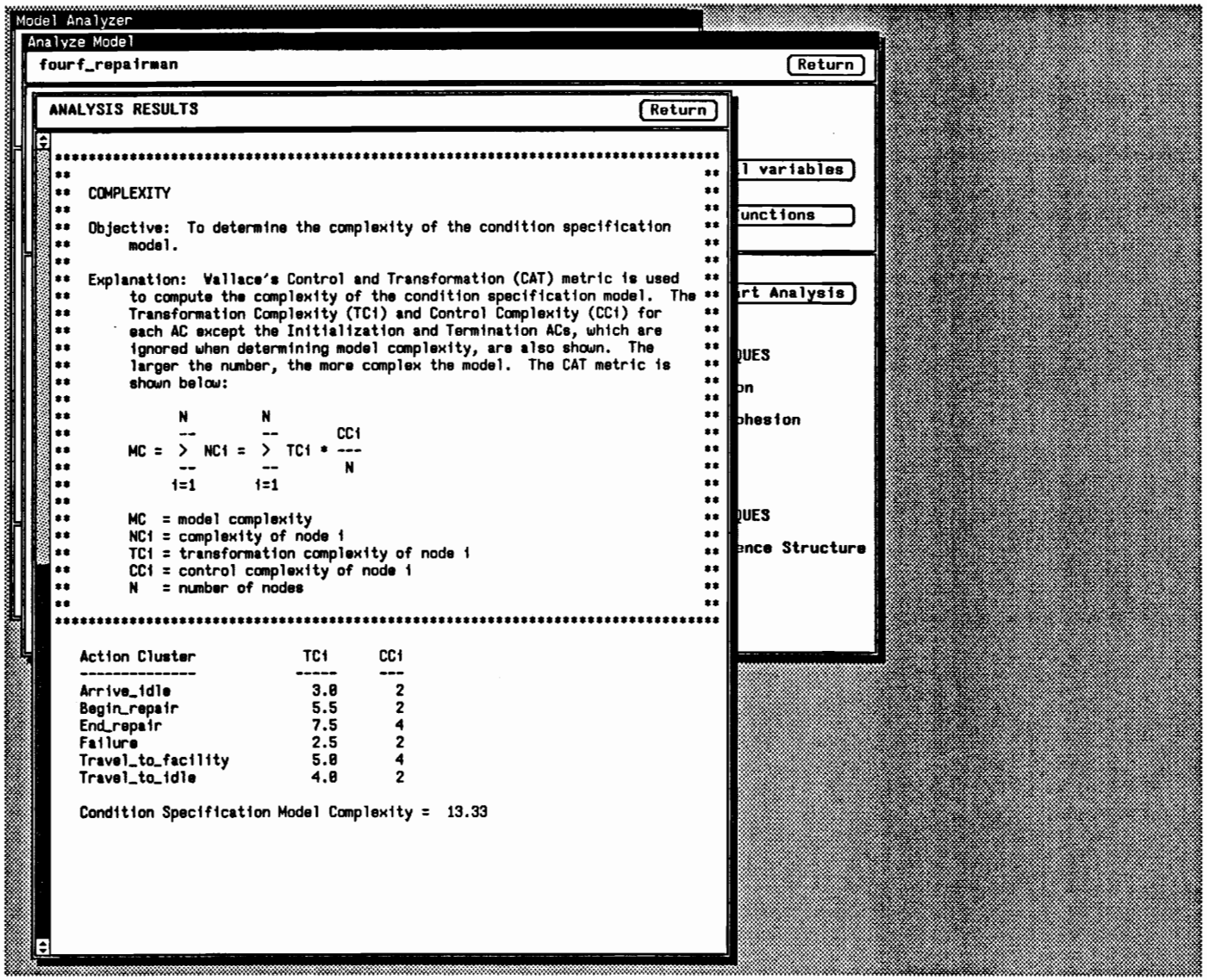


Figure 95. First Failed, First Fixed Machine Repairman Complexity Results

Table 15. Comparative Technique Results for Six Models

Model	Attribute Cohesion Index	Action Cluster Cohesion Index	Complexity Rating
Patrolling Repairman	6	5	8.70
Single Server Queue	3	3	12.83
Minimum Distance Machine Repairman	4	4	13.00
First Failed, First Fixed Machine Repairman	4	4	13.33
Automatic Inspection and Remote Unit	5	4	15.79
Harbor Model	4	3	26.00
AVERAGE	**4.3**	**3.8**	**14.94**
MEAN	**4**	**4**	**13.17**

comparative techniques for six models. In relationship to the six models, the First Failed, First Fixed Machine Repairman Model has the mean attribute and action cluster cohesion index and is slightly less than average in complexity.

6.3.3 Informative Techniques

The last diagnostic techniques applied to the model are the informative techniques. First, the immediate precedence structure test is performed, and the results are shown in Figure 96. Each action cluster in the First Failed, First Fixed Machine Repairman Model has either an immediate singular predecessor, an immediate singular successor, or both. This signifies the model is highly sequential in nature.

The last analysis action performed is the informative technique of decomposition, which the modeler checks and then depresses the "Start Analysis" button. An alert panel, shown in Figure 97, asks the modeler to select the execution order priority between "Arrive_idle" and "Begin_repair". The modeler determines these two ACs may never occur simultaneously and thus selects "Equal Priority". Thirteen other selections of execution order priorities are made before the decomposition results are displayed, as shown in Figure 98. Of the fourteen priorities made, only four do not have equal priorities: (1) "Failure" has priority over "Arrive_idle", (2) "Failure" has priority over "End_repair", (3) "Failure" has priority over "Travel_to_facility", and (4) "Failure" has priority over "Travel_to_idle". The decomposition results show the model broken into seven subgraphs, with only one subgraph containing more than one AC.

After careful examination of the simplified ACIG, the modeler determines that only two of the four above priorities are needed: (1) "Failure" has priority over "Arrive_idle" and (2) "Failure has priority over "End_repair". With these two execution order priorities established, the "Travel_to_idle" and "Travel_to_facility" ACs never

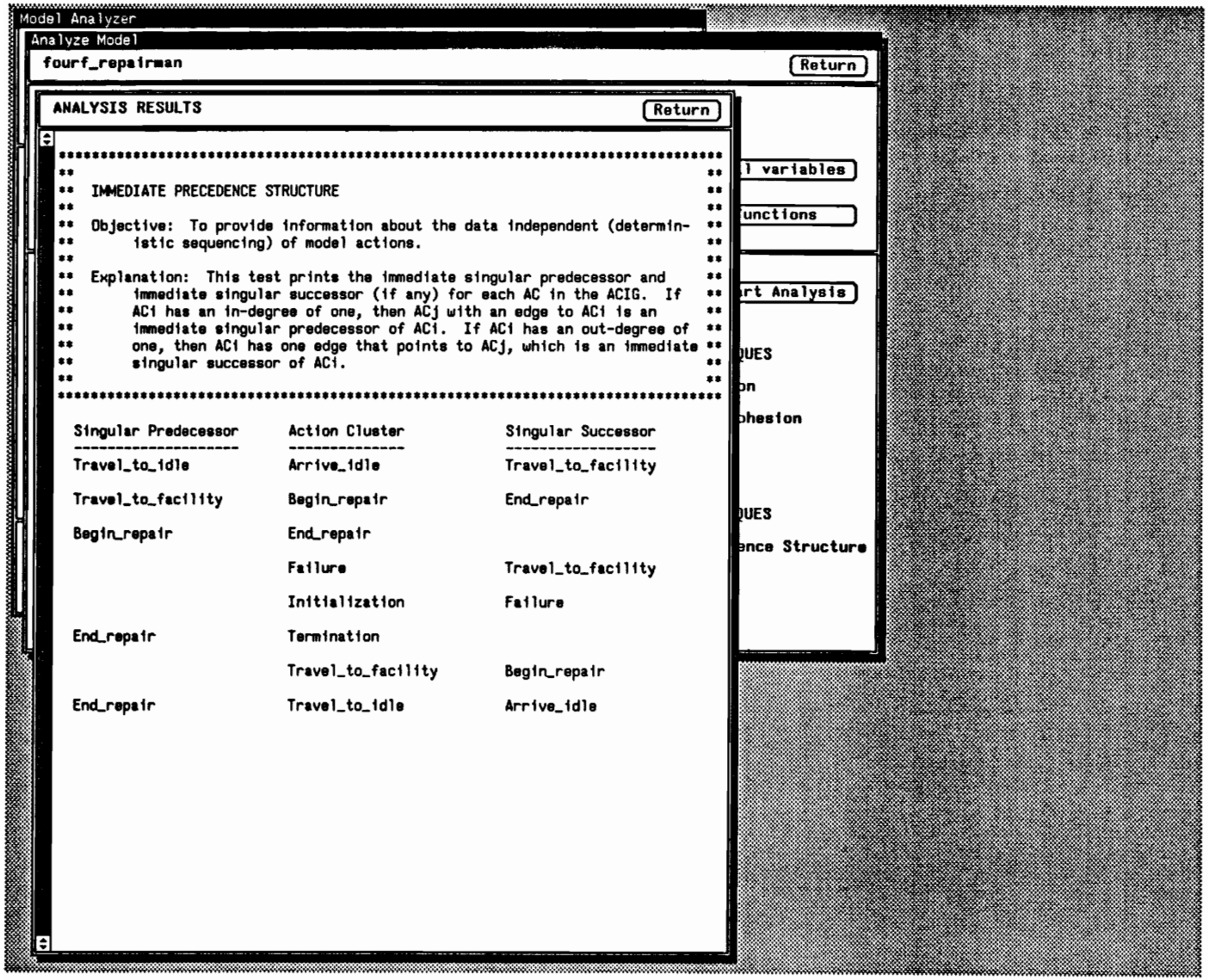


Figure 96. First Failed, First Fixed Machine Repairman Immediate Precedence Structure Results

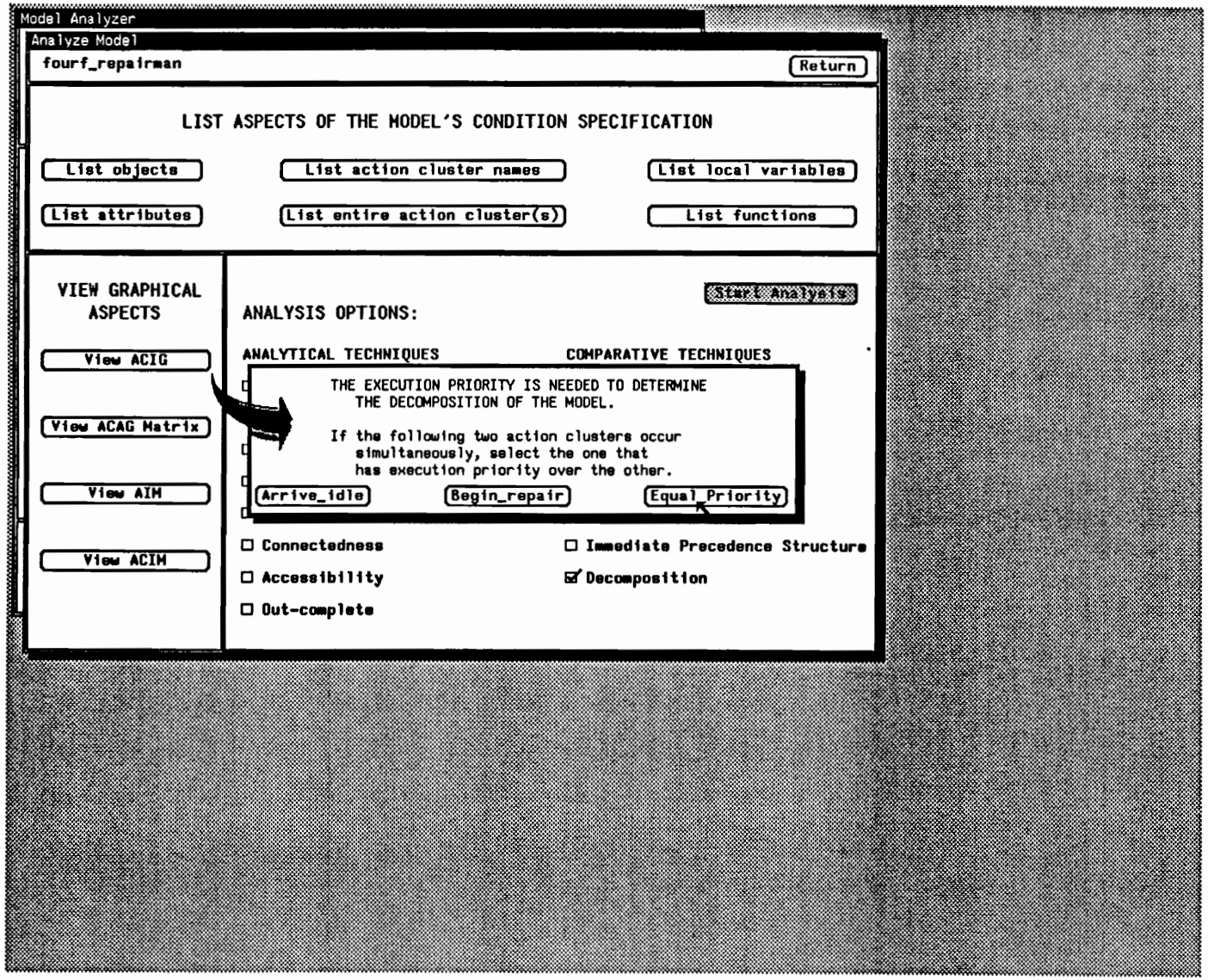


Figure 97. Alert Panel Used to Determine the Execution Order Priority Between "Arrive_idle" and "Begin_repair"

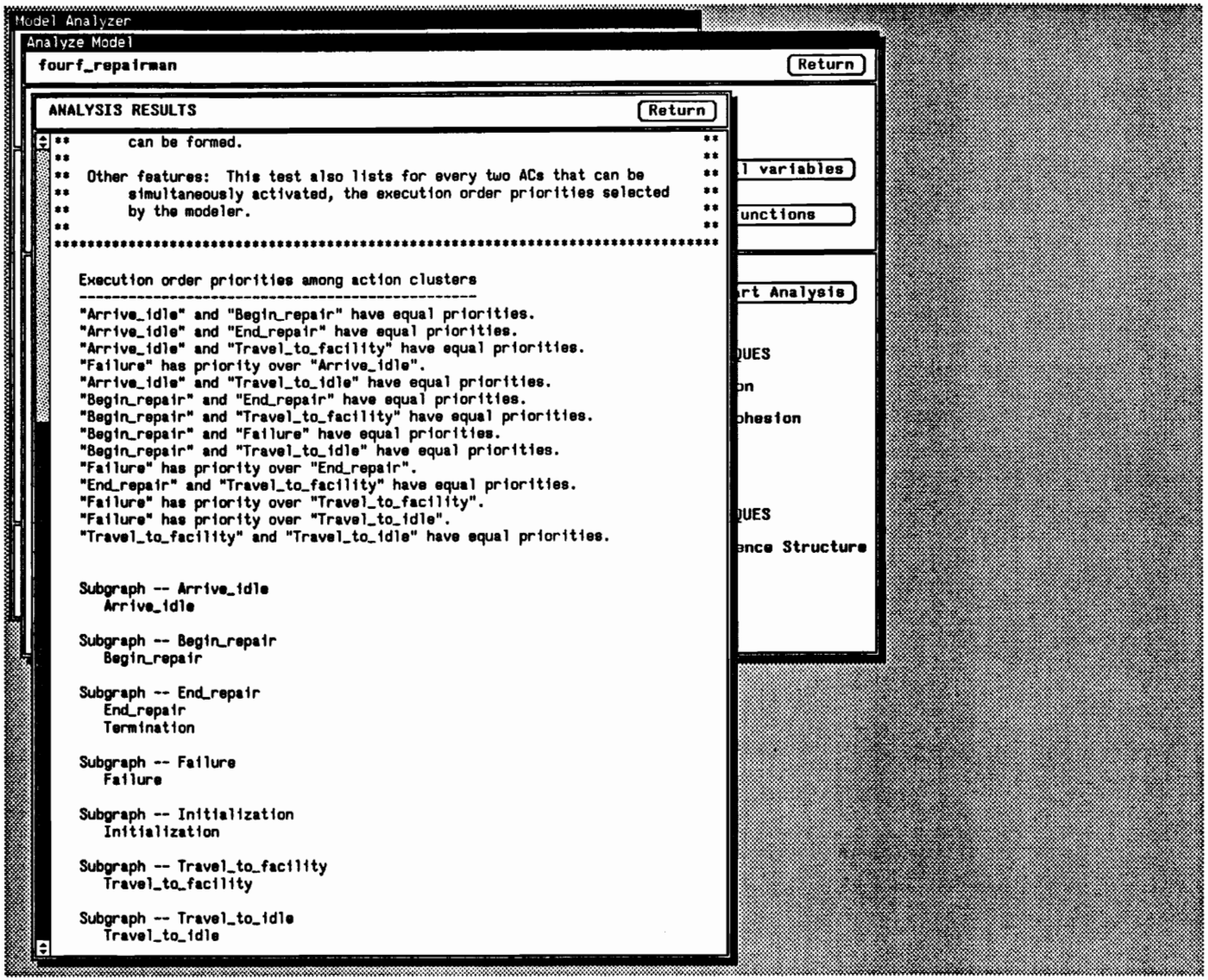


Figure 98. First Failed, First Fixed Machine Repairman Decomposition Results with Four Priorities Established

occur simultaneously with "Failure". The decomposition test is again performed on the model and this time only the two execution order priorities are established. The results, shown in Figure 99, now show five subgraphs. The decomposition of the model is highly dependent upon the modeler's selection of execution order priorities between action clusters.

The graphical version of the expanded ACIG consisting of these five subgraphs is shown in Figure 100, which clearly displays the sequential nature of the First Failed, First Fixed Machine Repairman Model. The super Condition Specification for the model, listed in Figures 101 and 102, is useful to the modeler during the translation of the specification into executable code.

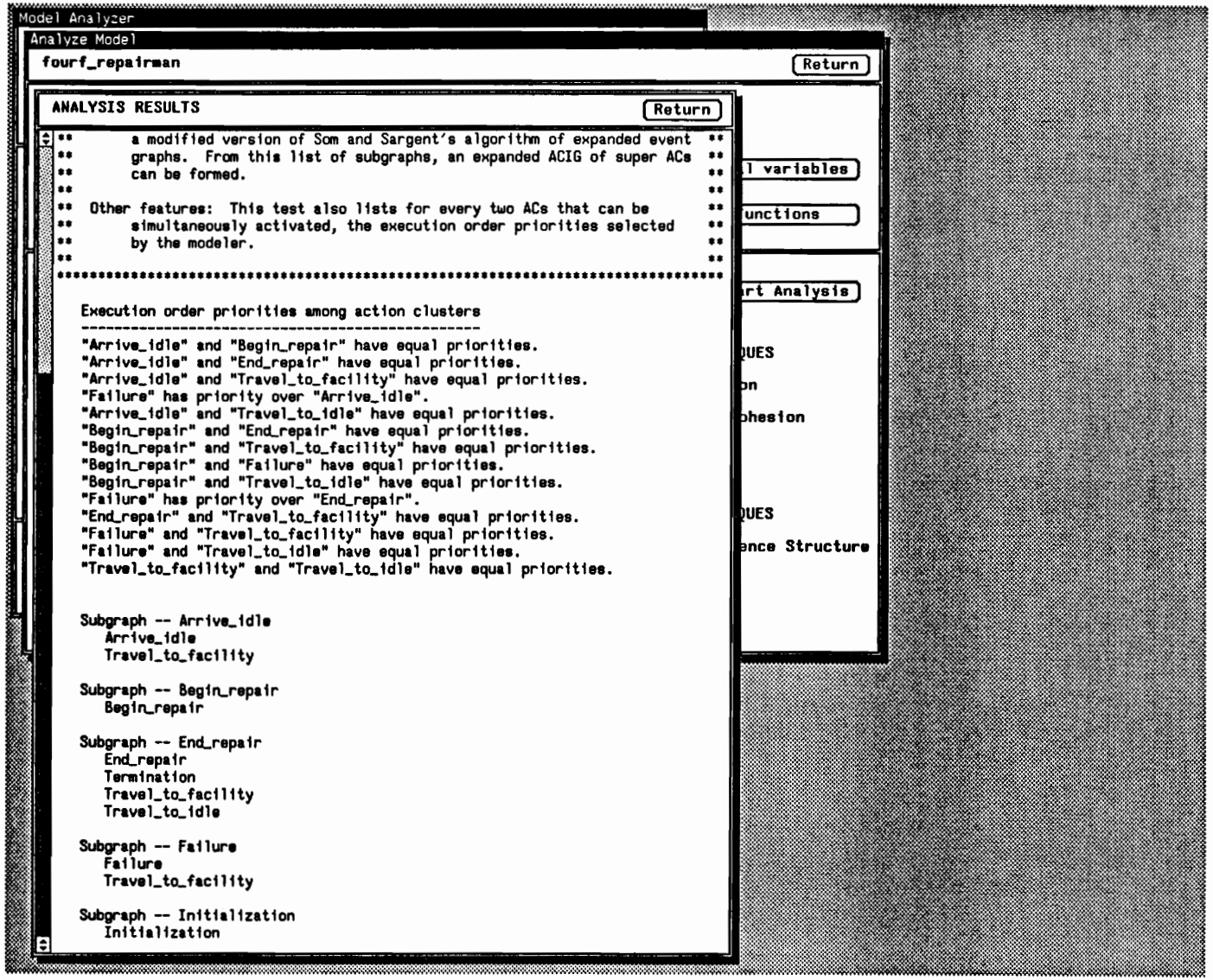


Figure 99. First Failed, First Fixed Machine Repairman Decomposition Results with Two Priorities Established

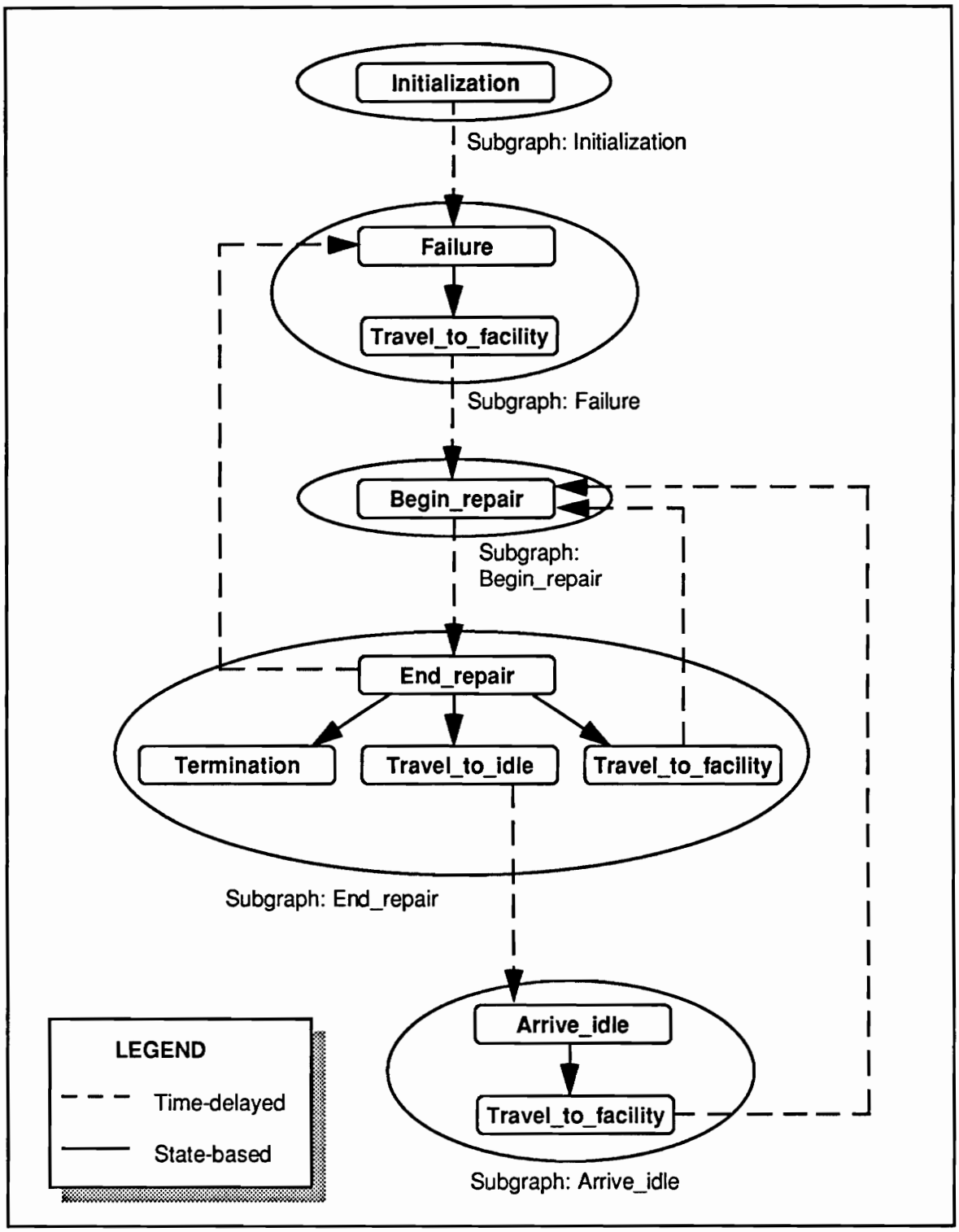


Figure 100. First Failed, First Fixed Machine Repairman Expanded Action Cluster Incidence Graph

```
Super AC: Initialization
    WHEN START:
        INPUT (mrp.n, mrp.max_repairs, mrp.mean_uptime, mrp.mean_repairtime);
        CREATE (repairman);
        FOR i := 1 TO mrp.n DO
            CREATE (facility[i]);
            facility[i].failed := FALSE;
            SET ALARM (mrp.failure, neg_exp(mrp.mean_uptime), i);
        END FOR;
        repairman.num_repairs := 0;
        repairman.location := idle;
        repairman.r_status := avail;

Super AC: Failure
    WHEN ALARM (mrp.failure, i):
        facility[i].failed := TRUE;
        Qinsert(mrp.failq, i);
        IF repairman.r_status = avail THEN
            SET ALARM (mrp.arr_fac, traveltime(repairman.location,
                fac[Qfirst(mrp.failq)]), i);
            Qdelete(mrp.failq);
            repairman.r_status := travel;
        END IF;

Super AC: Begin_repair
    WHEN ALARM (mrp.arr_fac, i):
        SET ALARM (mrp.end_repair, neg_exp(mrp.mean_repairtime), i);
        repairman.r_status := busy;
        repairman.location := fac[i];
```

Figure 101. Part One of First Failed, First Fixed Machine Repairman Super Condition Specification

```
Super AC: End_repair
    WHEN ALARM (mrp.end_repair, i):
        SET ALARM (mrp.failure, neg_exp(mrp.mean_uptime), i);
        facility[i].failed := FALSE;
        repairman.num_repairs := repairman.num_repairs + 1;
        IF repairman.num_repairs >= mrp.max_repairs THEN
            STOP;
        END IF;
        IF (FOR ALL i:1..mrp.n, NOT facility[i].failed)  THEN
            SET ALARM (mrp.arr_idle, traveltime(repairman.location, idle));
        ELSE
            SET ALARM (mrp.arr_fac, traveltime(repairman.location,
                fac[Qfirst(mrp.failq)]), i);
            Qdelete(mrp.failq);
        END IF;
        repairman.r_status := travel;

Super AC: Arrive_idle
        WHEN ALARM (mrp.arr_idle):
        repairman.r_status := avail;
        repairman.location := idle;
        IF (FOR SOME i:1..mrp.n, facility[i].failed) THEN
            SET ALARM (mrp.arr_fac, traveltime(repairman.location,
                fac[Qfirst(mrp.failq)]), i);
            Qdelete(mrp.failq);
            repairman.r_status := travel;
        END IF;
```

Figure 102. Part Two of First Failed, First Fixed Machine Repairman Super Condition Specification

7. Evaluation of Model Analyzer

The complete evaluation of a prototype requires time for conducting many tests and gathering data and feedback from experienced users. Since the Model Analyzer is recently developed, time does not permit a complete evaluation to this date; however, some testing results are compiled. The Model Analyzer analyzes the six different models, listed in Appendix C, and successfully produces accurate and informative analysis results for each model.

Determining if the design objectives of the Model Analyzer are accomplished is another means of evaluating the Model Analyzer. Below is an evaluation of how well the Model Analyzer meets each objective, outlined in Section 3.3.

(1) **Provide automated and semi-automated support to the modeler** — The Model Analyzer provides much automated support to the modeler. Some examples of automated support include automatically parsing the CS into lists; developing the unsimplified ACIG, ACAG, AIM, and ACIM; and performing analytical, comparative, and informative diagnosis on the model. During the informative decomposition test, the building of the execution order priority matrix requires user interaction, which is considered semi-automated. The only manual operation in the Model Analyzer is simplifying the ACIG, in which the modeler determines which infeasible edges are to be removed from the graph. This manual process is automated by the expert system described in Chapter 6, but the expert system is not integrated with the Model Analyzer.

(2) **Perform diagnostic analysis on non-executable model specifications, called Condition Specifications** — The input of the Model Analyzer is the Condition Specification for a model, and all diagnosis is performed upon this input.

(3) **Perform analytical, comparative, and informative diagnosis on both static and dynamic aspects of the model** — The analytical diagnostic techniques performed by the Analyzer are attribute utilization, attribute initialization, attribute reference, action cluster determinacy, statement order dependency, connectedness,

accessibility, and out-completeness. The Model Analyzer also implements the comparative techniques of attribute cohesion, action cluster cohesion, and complexity, and the informative techniques of attribute classification, immediate precedence structure, and decomposition. These diagnostic techniques are determined by the static information found in the object specification of the CS and the dynamic information listed in the transition specification.

(4) **Use graph-based diagnostic techniques** — The Model Analyzer employs the graph-based diagnostic techniques developed by Nance and Overstreet. Almost all of the diagnosis is gathered from two directed graphs: ACIG and ACAG.

(5) **Provide the user with graphical representations of the model** — The Model Analyzer graphically represents the model by displaying the ACIG in simplified and unsimplified forms as a matrix, circular graph, and linear graph; the ACAG as a matrix; and the AIM and ACIM as matrices.

(6) **Interact with the user through a graphical user interface** — The Model Analyzer is developed using SunView, which allows the development of a graphical user interface. Shown in the figures throughout this thesis, the interface contains canvases, panels, text subwindows, icons, buttons, multiple choice items, scrollbars, menus, and alerts. This graphical user interface allows the user to interact with the Model Analyzer with simple "point and click" mouse movements.

(7) **Use a relational database to store the CS and other information about the model** — INGRES, a relational database, is used by the Model Analyzer to store the CS and analysis information related to the model. As the input CS of a model is parsed, the parts of the CS and other information are stored in relations. After the CS is parsed, the input CS is never again accessed, since all the necessary information needed by the Model Analyzer is stored in organized tables (relations).

8. Summary and Conclusions

This research accomplishes its goal of building a Model Analyzer prototype, which provides early feedback in the model development life-cycle by analyzing the model specifications. The Model Analyzer specifically analyzes the Condition Specification, which is a world-view independent specification language, of a model. The analysis of the model specifications allows errors to be corrected before the translation of the model into executable code and provides information helpful in the later stages of verification, validation, and translation.

The Model Analyzer takes a file that contains the CS of a model as its input. This CS is parsed using Lex and Yacc, and its information is stored in relations of the INGRES relational database. Graph-based diagnostics, which are based upon the ACIG and ACAG, are performed on the model specifications. The Model Analyzer can list various aspects of the model, graphically view four different representations of the model, and perform analytical, comparative, and informative diagnostic assistance.

Although not included in the Model Analyzer, the research also develops a prototype expert system, which automatically simplifies the ACIG. By applying generic simulation knowledge to the CS of a model with the help of a large rule base and pattern matching, the expert system removes the infeasible edges. The output list of ACIG edges is not guaranteed to produce a totally simplified ACIG, but the expert system does produce a more simple ACIG than using Nance and Overstreet's approach and removes most of the infeasible edges.

Section 8.1 presents the accomplishments of the research and a conclusion. The final section discusses possible future research and improvements to be made to the Model Analyzer.

8.1 Conclusions

This research produces a Model Analyzer prototype with greater functionality than the previous prototype. The new prototype implements twelve of the fourteen diagnostic analysis techniques listed in Table 5. Of these twelve techniques, five have improved implementations: (1) initialization, (2) accessibility, (3) action cluster cohesion, (4) attribute cohesion, and (5) immediate precedence structure. Another technique, decomposition, implements a new concept, called the expanded ACIG, to decompose the model into subgraphs (super ACs). Two other new diagnostic techniques are developed: (1) attribute reference and (2) statement order determinacy.

Besides the new and modified analysis techniques, the new prototype graphically views four different model representations: (1) ACIG, (2) ACAG, (3) AIM, and (4) ACIM. The previous prototype only graphically viewed the ACIG. Also not included in the previous prototype is the use of a graphical user interface and relational database. The graphical user interface makes the prototype easier to use for the modeler, and the relational database makes it easier to analyze the information stored in the specification of the model.

Another important accomplishment is the use of an expert system to produce the simplified ACIG automatically. Together, the expert system and the Model Analyzer provide useful tools for analyzing the model specifications and strengthening the claim that using graph-based diagnosis with the Condition Specification is a very effective analysis technique.

8.2 Future Research

No research is ever totally complete and improvements are always possible. The list of possible improvements to the Model Analyzer is presented below.

(1) Expand the CS syntax to handle set manipulation and the syntax for the function and report specification sections.

(2) Provide the capabilities for greater error detection in the CS to include type checking.

(3) Include a separate database for each model. The databases collect analysis information that are not lost when exiting the Model Analyzer.

(4) Use color to brighten the interface and enhance the drawing of the graphical representations. Currently, the Model Analyzer uses only the colors of black, white, and shades of gray.

(5) Incorporate the expert system described in Chapter 5 into the Model Analyzer so that the simplification of the ACIG is an automatic process.

(6) Provide the capability to print out the graphical representations and results of the analysis tests.

(7) Improve the diagnostic of statement order dependency to include the capability to detect when an input attribute is also used as an output attribute in the same statement (action).

(8) Improve the diagnostic of decomposition by making it easier to develop the execution order priorities among action clusters.

(9) Include diagnostics for attribute consistency and revision consistency and develop new diagnostics.

These improvements should prove useful in the definition of the next prototype; however, major advances in succeeding prototypes are likely to be based on technology

improvements that open new areas for human-computer interaction. Until these technology improvements are made, succeeding prototypes are not likely to make major advancements to the current prototype.

Bibliography

Aho, A.V., J.E. Hopcroft, and J.D. Ullman (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, MA.

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

Aho, A.V., R. Sethi, and J.D. Ullman (1986), *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, MA.

Balci, O. (1986), "Requirements for Model Development Environments," *Computers & Operations Research 13*, 1, 53-67.

Balci, O. (1987a), "Guidelines for Successful Simulation Studies, Part I," Technical Report, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Balci, O. (1987b), "Guidelines for Successful Simulation Studies, Part II," Technical Report, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Balci, O. (1988), "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages," In *Proceedings of the 1988 Winter Simulation Conference,* M. Abrams, P. Haigh, and J. Comfort, Eds. IEEE, Piscataway, NJ, 287-295.

Balci, O. (1990), "Guidelines for Successful Simulation Studies," In *Proceedings of the 1990 Winter Simulation Conference,* O. Balci, R.P. Sadowski, and R.E. Nance, Eds. IEEE, Piscataway, NJ, 25-32.

Balci, O. and R.E. Nance (1987a), "Simulation Model Development Environments: A Research Prototype," *Journal of the Operational Research Society 38*, 8, 753-763.

Balci, O. and R.E. Nance (1987b), "Simulation Support: Prototyping the Automation-Based Paradigm," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 495-502.

Balci, O., R.E. Nance, E.J. Derrick, E.H. Page, and J.L. Bishop (1990), "Model Generation Issues in a Simulation Support Environment," In *Proceedings of the 1990 Winter Simulation Conference,* O. Balci, R.P. Sadowski, and R.E. Nance, Eds. IEEE, Piscataway, NJ, 257-263.

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

Barr, A. and E.A. Feigenbaum, Eds. (1981), *The Handbook of Artificial Intelligence, Volume I*, Addison-Wesley Publishing Company, Reading, MA.

Barr, A. and E.A. Feigenbaum, Eds. (1982), *The Handbook of Artificial Intelligence, Volume II*, Addison-Wesley Publishing Company, Reading, MA.

Barr, A., P.R. Cohen, and E.A. Feigenbaum, Eds. (1989), *The Handbook of Artificial Intelligence, Volume IV*, Addison-Wesley Publishing Company, Reading, MA.

Bauer, K.W., B. Kochar, and J.J. Talavage (1985), "Simulation Model Decomposition by Factor Analysis," In *Proceedings of the 1985 Winter Simulation Conference*, D.T. Gantz, G.C. Blais, and S.L. Solomon, Eds. IEEE, Piscataway, NJ, 185-188.

Bauer, K.W., Jr., B. Kochar, and J.J. Talavage (1991), "Discrete Event Simulation Model Decomposition by Principal Components Analysis," *ORSA Journal on Computing 3*, 1, 23-32.

Beams, J. (1988), "Diagnosis of Digraph Representations of Simulation Models," CS 5160 Project, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Berzins, V. (1985), "Analysis and Design in MSG.84," *IEEE Transactions on Software Engineering SE-11*, 8, 657-670.

Blum, B.I. (1982), "The Life Cycle — A Debate Over Alternate Models," *ACM SIGSOFT Software Engineering Notes 7*, 4, 18-20.

Boehm, B.W. (1986), "A Spiral Model of Software Development and Enhancement," *ACM Software Engineering Notes 11*,4, 14-24. Also available in *Computer 21*, 5 (1988), 61-72.

Buchanan, B.G. (1982), "New Research on Expert Systems," *Machine Intelligence, Vol. 10*, J.E. Hayes, D. Michie, and Y-H Pao, Eds. Ellis Horwood Limited, Chichester, Sussex, England, 269-299.

Buchanan, B.G. and R.O. Duda (1983), "Principles of Rule-Based Expert Systems," *Advances in Computers 22*, 163-216.

Burns, J.R. and W.H. Winstead (1985), "M-Labeled Digraphs: An Aid to the Use of Structural and Simulation Models," *Management Science 31*, 3, 343-357.

Carne, E.B. (1965), *Artificial Intelligence Techniques*, MacMillan and Company, Ltd., London, England.

Charniak, E. and D. McDermott (1985), *Introduction of Artificial Intelligence*, Addison-Wesley Publishing Company, Reading, MA.

Clancey, W.J. (1983), "The Epistemology of a Rule-Based Expert System — A Framework for Explanation," *Artificial Intelligence 20*, 215-251.

Clark, K.L. and F.G. McCabe (1982), "PROLOG: A Language for Implementing Expert Systems," *Machine Intelligence, Vol. 10*, J.E. Hayes, D. Michie, and Y-H Pao, Eds. Ellis Horwood Limited, Chichester, Sussex, England, 455-475.

Cota, B.A. and R.G. Sargent (1990a), "A New Version of the Process World View for Simulation Modeling," Technical Report 9003, CASE Center, Syracuse University, Syracuse, NY.

Cota, B.A. and R.G. Sargent (1990b), "Simultaneous Events and Distributed Simulation", In *Proceedings of the 1990 Winter Simulation Conference,* O. Balci, R.P. Sadowski, and R.E. Nance, Eds. IEEE, Piscataway, NJ, 436-440.

Davies, R.M. and R.M. O'Keefe (1989), *Simulation Modelling with Pascal,*Prentice Hall International (UK) Ltd., Hertfordshire, England.

Davis, R. and J. King (1977), "An Overview of Production Systems," *Machine Intelligence, Vol. 8*, E.W. Elcock and D. Michie, Eds. Ellis Horwood Limited, Chichester, Sussex, England, 300-332.

Deloria, D. (1987), "Practical Yacc - A Gentle Introduction to the Power of This Famous Parser Generator," *The C Users' Group Newsletter 5*, 2x, 18-25.

Department of Defense (1989), *Critical Technologies Plan for the Committees on Armed Services, United States Congress, Revised,* AD-A210 170, Department of Defense, Washington, DC.

Fishman, G.S. (1973), *Concepts and Methods in Discrete Event Digital Simulation,* John Wiley & Sons, New York, NY.

Frankel, V.L. (1987), "A Prototype Assistance Manager for the Simulation Model Development Environment," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Frankel, V.L. and O. Balci (1989), "A Prototype Assistance Manager for the Simulation Model Development Environment," Technical Report SRC-87-009, Systems Research Center, Virginia Tech, Blacksburg, VA.

Frankel, V.L. and O. Balci (1989), "An On-Line Assistance System for the Simulation Model Development Environment," *International Journal of Man-Machine Studies 31*, 699-716.

Freeman, P. (1983), "Fundamentals of Design," In *Tutorial on Software Design,* P. Freeman and A. Wasserman, Eds. IEEE Computer Society Press, Piscataway, NJ, 2-22.

Gevarter, W.B. (1984), *Artificial Intelligence, Expert Systems, Computer Vision, and Natural Language Processing*, Noyes Publications, Park Ridge, NJ.

Hartson, H.R. and D. Hix (1989), "Human-Computer Interface Development: Concepts and Systems for Its Management," *ACM Computing Surveys 21*, 1, 5-92.

Hartson, H.R., A.C. Siochi, and D. Hix (1990), "The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs," *ACM Transactions on Information Systems 8,* 3, 181-203.

Hix, D. and R.S. Schulman (1991), "Human-Computer Interface Development Tools: A Methodology for Their Evaluation," *Communications of the ACM 34*, 3, 74-87.

Horowitz, B.M. (1990), "Defence Science Board Recommendations: An Examination of Defence Policy on the Use of Modeling and Simulation," In *Proceedings of the 1990 Winter Simulation Conference,* O. Balci, R.P. Sadowski, and R.E. Nance, Eds. IEEE, Piscataway, NJ, 224-230.

Johnson, S.C. (1983), "Yacc: Yet Another Compiler-Compiler," In *The UNIX Programmer's Manual, Volume 2*, Bell Telephone Laboratories, Inc., Murray Hill, NJ.

Kernighan, B.W. and R. Pike (1984), *The UNIX Programming Environment*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Lesk, M.E. and E. Schmidt (1983), "Lex - A Lexical Analyzer Generator," In *The UNIX Programmer's Manual, Volume 2*, Bell Telephone Laboratories, Inc., Murray Hill, NJ.

Manber, U. (1989), *Introduction to Algorithms A Creative Approach,* Addison-Wesley Publishing Company Inc., Reading, MA.

Moose, R.L. Jr. and R.E. Nance (1987), "Model Analysis in a Model Development Environment," Technical Report SRC-87-010, Systems Research Center, Virginia Tech, Blacksburg, VA.

Moose, R.L. Jr. and R.E. Nance (1989), "The Design and Development of an Analyzer for Discrete Event Model Specifications," In *Impacts of Recent Computer Advances on Operations Research*, R. Sharda, B.L. Golden, E. Wasil, O. Balci, and W. Stewart, Eds. Elsevier Science Publishing, New York, NY, 407-421.

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

Nance, R.E. (1981b), "The Time and State Relationships in Simulation Modeling," *Communications of the ACM 24*, 4, 173-179.

Nance, R.E. (1984), "Model Development Revisited," In *Proceedings of the 1984 Winter Simulation Conference*, S. Sheppard, U.W. Pooch, and C.D. Pegden, Eds. IEEE, Piscataway, NJ, 75-80.

Nance, R.E. (1987), "The Conical Methodology: A Framework for Simulation Model Development," In *Proceedings of the Conference on Methodology and Validation,* O. Balci, Ed. SCS, San Diego, CA, 38-43.

Nance, R.E. and J.D. Arthur (1988), "The Methodology Roles in the Realization of a Model Development Environment," In *Proceedings of the 1988 Winter Simulation Conference*, M. Abrams, P. Haigh, and J. Comfort, Eds. IEEE, Piscataway, NJ, 220-225.

Nance, R.E., R.L. Moose, Jr., and O. Balci (1984), "Evaluation of the UNIX Host for a Model Development Environment," In *Proceedings of the 1984 Winter Simulation Conference*, S. Sheppard, U.W. Pooch, and C.D. Pegden, Eds. IEEE, Piscataway, NJ, 577-584.

Nance, R.E. and C.M. Overstreet (1986), "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," Technical Report SRC-86-001, Systems Research Center, Virginia Tech, Blacksburg, VA.

Nance, R.E. and C.M. Overstreet (1987a), "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," *Transactions of the Society for Computer Simulation 4*, 1, 33-57.

Nance, R.E. and C.M. Overstreet (1987b), "Exploring the Forms of Model Diagnosis in a Simulation Support Environment," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 590-596.

Neelamkavil, F. (1987), *Computer Simulation and Modelling,* John Wiley & Sons Ltd., New York, NY.

Overstreet, C.M. (1982), "Model Specification and Analysis for Discrete Event Simulation," Ph.D. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Overstreet, C.M. (1987), "Using Graphs to Translate Between World Views," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 582-589.

Overstreet, C.M. and R.E. Nance (1984), "Graph-Based Diagnosis of Discrete Event Model Specifications," Technical Report CS83028-R, Revised Draft, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Overstreet, C.M. and R.E. Nance (1985), "A Specification Language to Assist in Analysis of Discrete Event Simulation Models," *Communications of the ACM 28*, 2, 190-201.

Overstreet, C.M. and R.E. Nance (1986), "World View Based Discrete Event Model Simplification," In *Modelling and Simulation Methodology in the Artificial Intelligence Area*, M.S. Elzas, T.I. Ören, and B.P. Zeigler, Eds. Elsevier Science Publishers, Amsterdam, The Netherlands, 165-179.

Overstreet, C.M., R.E. Nance, O. Balci, and L.F. Barger (1986), "Specification Languages: Understanding Their Role in Simulation Model Development," Technical Report SRC-87-001, Systems Research Center, Virginia Tech, Blacksburg, VA.

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

Peterson, J.L. (1977), "Petri Nets," *Computing Surveys 9*, 3, 223-252.

Pressman, R.S. (1987), *Software Engineering: A Practitioner's Approach*, Second Edition, McGraw-Hill, New York, NY, 3-11.

Quintus Computer Systems (1990a), *Quintus Prolog Reference Manual*, Quintus Computer Systems, Inc., Mountain View, CA.

Quintus Computer Systems (1990b), *Quintus Prolog User's Guide*, Quintus Computer Systems, Inc., Mountain View, CA.

Rowe, L.A., M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan (1987), "A Browser for Directed Graphs," *Software - Practice and Experience 17*, 1, 61-76.

Sargent, R.G. (1988), "Event Graph Modelling for Simulation with an Application to Flexible Manufacturing Systems," *Management Science 34*, 10, 1231-1251.

Schruben, L.W. (1983), "Simulation Modeling with Event Graphs," *Communications of the ACM 26*, 11, 957-963.

Schruben, L.W. (1987), "A "Disposable" Graphical Event Synthesizer for Teaching Simulation Model Building," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 72-76.

Schruben, L.W. (1990), "Simulation Graphical Modeling and Analysis (SIGMA) Tutorial," In *Proceedings of the 1990 Winter Simulation Conference,* O. Balci, R.P. Sadowski, and R.E. Nance, Eds. IEEE, Piscataway, NJ, 158-161.

Schruben, L.W. and D. Briskman (1988), "Teaching Simulation with S," In *Proceedings of the 1988 Winter Simulation Conference,* M. Abrams, P. Haigh, and J. Comfort, Eds. IEEE, Piscataway, NJ, 869-874.

Schruben, L.W. and E. Yucesan (1988), "Simulation Graphs," In *Proceedings of the 1988 Winter Simulation Conference,* M. Abrams, P. Haigh, and J. Comfort, Eds. IEEE, Piscataway, NJ, 504-508.

Schruben, L.W. and E. Yucesan (1989), "Simulation Graph Duality: A World View Transformations for Simple Queueing Models," In *Proceedings of the 1989 Winter Simulation Conference,* E.A. MacNair, K.J. Musselman, P. Heidelberger, Eds. IEEE, Piscataway, NJ, 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 1*, 2, 107-125.

Stefik, M., J. Aikins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti (1982), "The Organization of Expert Systems, A Tutorial," *Artificial Intelligence 18*, 135-173.

Stoegerer, J.K. (1984), "A Comprehensive Approach to Specification Languages," *The Australian Computer Journal 16*, 1, 1-13.

Stonebraker, M., E. Wong, and P. Kreps (1976), "The Design and Implementation of INGRES," *ACM Transactions on Database Systems 1,* 3, 189-222.

Sun Microsystems (1985), *Introduction to SunINGRES*, Revision C, Sun Microsystems, Inc., Mountain View, CA.

Sun Microsystems (1987), *SunINGRES Manual Set*, Release 5.0, Revision A, Sun Microsystems, Inc., Mountain View, CA.

Sun Microsystems (1988a), *Programming Utilities & Libraries*, Revision A, Sun Microsystems, Inc., Mountain View, CA.

Sun Microsystems (1988b), *SunView 1 Beginner's Guide*, Revision A, Sun Microsystems, Inc., Mountain View, CA.

Sun Microsystems (1988c), *SunView 1 Programmer's Guide*, Revision A, Sun Microsystems, Inc., Mountain View, CA.

Thesen, A. and L.E. Travis (1990), "Introduction to Simulation," In *Proceedings of the 1990 Winter Simulation Conference,* O. Balci, R.P. Sadowski, and R.E. Nance, Eds. IEEE, Piscataway, NJ, 14-21.

Volkman, V. (1987), "Software Tools for Compiler Development," *The C Users' Group Newsletter 5*, 2, 12-25.

Volkman, V. (1987), "Yacc & Lex: Software Tools for Parser Design and Code Generation," *The C Users' Group Newsletter 5*, 2x, 30-41.

Wallace, J.C. (1985), "The Control and Transformational Metric: A Basis for Measuring Model Complexity," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Wallace, J.C. and R.E. Nance (1985), "The Control and Transformational Metric: A Basis for Measuring Model Complexity," Technical Report TR-85-15, Department of Computer Science, Virginia Tech, Blacksburg, VA.

Wallace, J.C. (1987), "The Control and Transformational Metric: Toward the Measurement of Simulation Model Complexity," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds., IEEE, Piscataway, NJ, 597-603.

Wasserman, A.I. (1983), "Information System Design Methodology," In *Tutorial on Software Design,* P. Freeman and A. Wasserman, Eds. IEEE Computer Society Press, Piscataway, NJ, 43-62.

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

Zave, P. (1984a), "The Operational Versus the Conventional Approach to Software Development," *Communications of the ACM 27,* 2, 104-118.

Zave, P. (1984b), "An Overview of the PAISLey Project — 1984," *ACM SIGSOFT Software Engineering Notes 9*, 4, 12-19.

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

Zeigler, B.P. (1984), *Multifacetted Modelling and Discrete Event Simulation*, Academic Press Inc., Orlando, FL.

Appendix A. Model Analyzer Interface Specification Diagrams in User Action Notation

Task: Select_Model			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[model_name] M_Lv	model_name!	selected = model_name	
M_L^	model_name-!, display(model_name) in frame to right of "Model Name:"		Model becomes current (selected) model

Figure 103. UAN: Select_Model

Task: Exit_Analyzer			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[stop_icon] $M_L \vee \wedge$	stop_icon!-!, display(quit_alert)		
~[confirm_button in quit_alert] $M_L \vee \wedge$	erase(quit_alert), erase(model_analyzer_frame)		Model Analyzer is terminated

Figure 104. UAN: Exit_Analyzer

Task: View_Model_CS			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
Select_Model		selected = model	
~[eye_icon] M_L v^	eye_icon!-!, display(model_viewer_frame), frame contains CS file of selected model		execute view_file CS file is opened read only

Figure 105. UAN: View_Model_CS

Task: Edit_Model_CS			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
View_Model_CS			CS file is opened as read only
~[edit_button] M_L∨∧	edit_button!-!, display(▲ editing cursor in CS file)		CS file is read/write
Perform editing actions	display(editing changes)		
~[quit/no_save_button] M_L∨∧ OR ~[return/save_button] M_L∨∧	quit/no_save_button!-!, erase(model_viewer_frame) OR return_save_button!-!, erase(model_viewer_frame)		Edited file is deleted and CS file is same as before editing. Edited file replaces the old file.

Figure 106. UAN: Edit_Model_CS

Task: Analyze_Model			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
Select_Model		selected = model	
~[magnifying_glass_ icon] M_{L}∨∧	magnifying_glass_icon!-!, cursor changes to hourglass, display(wait_message), parsing error: change cursor back to arrow, erase(wait_message), display(error_message) no parsing error: change cursor back to arrow, erase (wait_message), display(analyze_model_frame)		Parsing CS Model Parsing Halts Model is prepared to be analyzed

Figure 107. UAN: Analyze_Model

Task: Quit_Analysis_Session			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[return_button in analyze_model_frame] M_Lv^	return_button!-!, display(stop_analysis_alert)		
~[confirm_button in stop_analysis_alert] M_Lv^	erase(stop_analysis_alert), erase(analyze_model_frame)		Quit analysis of model

Figure 108. UAN: Quit_Analysis_Session

Task: List_Model_Objects			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[list_objects_button in analyze_model_frame] M_{L}^^	list_objects_button!-!, display(objects_list_panel)		make_objects_list performed

Figure 109. UAN: List_Model_Objects

Task: List_Action_Cluster_Names			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[list_ac_names_button in analyze_model_ frame] M_{L} v^	list_ac_names_button!-!, display(ac_list_panel)		execute make_ac_list

Figure 110. UAN: List_Action_Cluster_Names

Task: Pull_right(X_item in Y_menu)			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[X_item in Y_menu]	X_item!		
~[⇒ in X_item]	display(Z_menu)	selected = X_item	

Figure 111. UAN: Pull_right(X_item in Y_menu)

Task: List_Attributes_of_Objects			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[list_attributes_button in analyze_model_ frame] $M_L \vee \wedge$	list_attributes_button!-!, display(attributes_menu)		
Pull_right(Of_Object(s) in attributes_menu)	Of_Object(s)!, display(attribute_type_menu)	selected = Of_Object(s)	
Pull_right(type' in attribute_type_menu)	type'!, display(for_object_menu)	selected_type = type'	
~[object' in for_object_menu] $M_L \vee \wedge$	object'!, erase(for_object_menu), erase(attribute_type_menu), erase(attributes_menu),	selected_object = object'	
	selected_type = All Types: display(object_attribute_list_ panel)		if selected_object = All Objects make_obj_att_ all_list, else make_obj_att_list
	selected_type = Indicative: display(indicative_attribute_ list_panel)		if selected_object = All Objects make_ind_att_ all_list, else make_ind_att_list
	selected_type = Relational: display(relational_attribute_ list_panel)		if selected_object = All Objects make_rel_att_ all_list, else make_rel_att_list

Figure 112. UAN: List_Attributes_of_Objects

Task: List_Attribute_Classification_in_Action_Clusters

User Actions	Interface Feedback	Interface State	Connection To Computation
~[list_attributes_button in analyze_model_ frame] M_L∨∧	list_attributes_button!-!, display(attributes_menu)		
Pull_right(Within_ Action_Clusters(s) in attributes_menu)	Within_Action_Cluster(s)!, display(attribute_type_menu)	selected = Within_Action_ Cluster(s)	
Pull_right(type' in attribute_type_menu)	type'!, display(for_ac_menu)	selected_type = type'	
~[ac' in for_ac_menu] M_L∨∧	ac'!, erase(for_ac_menu), erase(attribute_type_menu), erase(attributes_menu),	selected_ac = ac'	
	selected_type = All Types: display(all_attributes_list_ panel)		if selected_ac = All ACs make_all_atts_ all_list, else make_all_atts_list
	selected_type = Input: display(input_attribute_ list_panel)		if selected_object = All Objects make_in_atts_ all_list, else make_in_atts_list
	selected_type = Output: display(output_attribute_ list_panel)		if selected_object = All Objects make_out_atts_ all_list, else make_out_atts_list
	selected_type = Control: display(control_attribute_ list_panel)		if selected_object = All Objects make_con_atts_ all_list, else make_con_att_list

Figure 113. UAN: List_Attribute_Classification_in_Action_Clusters

Task: List_Contents_of_Action_Clusters			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[list_entire_ac_button in analyze_model_ frame] M_L∨∧	list_entire_ac_button!-!, display(for_ac_menu)		
~[ac' in for_ac_menu] M_L∨∧	ac'!, erase(for_ac_menu) display(entire_ac_list_panel)	selected = ac'	if selected = All Action Clusters make_entire_ac_ all_list else make_entire_ac_ list

Figure 114. UAN: List_Contents_of_Action_Clusters

Task: List_Local_Variables_in_Action_Clusters			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[list_local_variables_ button in analyze_ model_ frame] $M_L \vee \wedge$	list_local_variables_button!-!, display(for_ac_menu)		
~[ac' in for_ac_menu] $M_L \vee \wedge$	ac'!, erase(for_ac_menu) display(local_variable_list_ panel)	selected = ac'	if selected = All Action Clusters make_loc_var_ all_list else make_loc_var_ list

Figure 115. UAN: List_Local_Variables_in_Action_Clusters

Task: List_Functions_in_Action_Clusters			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[list_functions_button in analyze_ model_ frame] M_L^^	list_functions_button!-!, display(for_ac_menu)		
~[ac' in for_ac_menu] M_L^^	ac'!, erase(for_ac_menu) display(function_list_panel)	selected = ac'	if selected = All Action Clusters make_funct_all_ list else make_funct_list

Figure 116. UAN: List_Functions_in_Action_Clusters

Task: View_Unsimplified_ACIG			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[view_acig_button in analyze_ model_ frame] M_L^^	view_acig_button!-!, display(acig_type_menu)		
Pull_right(Unsimplified in acig_type_menu	Unsimplified!, display(acig_form_menu)	selected = Unsimplified	
~[form' in acig_form_menu] M_L^^	form'!, erase(acig_form_menu), erase(acig_type_menu), selected_form = Matrix: display(unsimplified_acig_ matrix) selected_form = Circular Graph: display(unsimplified_acig_ circular) selected_form = Linear Graph: display(unsimplified_acig_ linear)	selected_form = form'	 execute make_acig_matrix execute make_ acig_cir_graph execute make_ acig_lin_graph

Figure 117. UAN: View_Unsimplified_ACIG

Task: View_Simplified_ACIG			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[view_acig_button in analyze_ model_ frame] M_L∨∧	view_acig_button!-!, display(acig_type_menu)		
Pull_right(Simplified in acig_type_menu	Simplified!, display(acig_form_menu)	selected = Simplified	
~[form' in acig_form_menu] M_L∨∧	form'!, erase(acig_form_menu), erase(acig_type_menu), selected_form = Matrix: display(simplified_acig_ matrix) selected_form = Circular Graph: display(simplified_acig_ circular) selected_form = Linear Graph: display(simplified_acig_ linear)	selected_form = form'	 execute make_acig_matrix execute make_ acig_cir_graph execute make_ acig_lin_graph

Figure 118. UAN: View_Simplified_ACIG

Task: Simplify_ACIG			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
View_Simplified_ACIG			
Remove_Infeasible_ Edges			
~[redraw_graph_button in simplified_acig_ panel] $M_L \vee \wedge$	redraw_graph_button!-!, display(new simplified_acig_ panel), erase(old simplified_acig_ panel)		execute redraw procedure

Figure 119. UAN: Simplify_ACIG

Task: Remove_Infeasible_Edges			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[remove_edge(s)_ button in simplified_ acig_panel] $M_L \vee \wedge$	remove_edge(s)_button!-!, display(remove_acig_edges_ panel)		
([edge' in remove_acig_edges_ panel] $M_L \vee \wedge$)*	edge'!, display(new remove_acig_ edges_panel), erase(old remove_acig_edges_ panel)	selected = edge'	Remove edge' from Simplified ACIG relation in database
~[return_button in remove_acig_edges_ panel] $M_L \vee \wedge$	return_button!-!, erase(remove_acig_edges_ panel)		

Figure 120. UAN: Remove_Infeasible_Edges

Task: View_ACAG_Matrix			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[view_acag_matrix_ button in analyze_ model_frame] M_L∨∧	view_acag_matrix_button!-!, display(acag_matrix_panel)		execute make_acag_graph

Figure 121. UAN: View_ACAG_Matrix

Task: View_AIM			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[view_aim_button in analyze_model_frame] M_L^^	view_aim_button!-!, display(aim_panel)		execute make_aim

Figure 122. UAN: View_AIM

Task: View_ACIM			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
~[view_acim_button in analyze_model_frame] M_Lv^	view_acim_button!-!, display(acim_panel)		execute make_acim

Figure 123. UAN: View_ACIM

Task: Perform_Diagnostic_Techniques			
User Actions	**Interface Feedback**	**Interface State**	**Connection To Computation**
(~[technique' in analysis_options_panel] $M_L \vee \wedge$)*	no √ in ❑ next to technique': display(√ in ❑ next to technique') √ in ❑ next to technique': erase(√ in ❑ next to technique')	selected = technique' deselected = technique'	
~[start_analysis_button in analysis_options_ panel] $M_L \vee \wedge$	change cursor to hourglass, √ in ❑ next to Decomposition: change cursor back to arrow, display(execution_priority_ alert)		Perform selected techniques
{(~[priority_ac' in execution_priority_alert] $M_L \vee \wedge$)*}	(display(execution_priority_ alert))*, change cursor to hourglass,	selected_priority = priority_ac'	
	change cursor back to arrow, display(results_panel)		

Figure 124. UAN: Perform_Diagnostic_Techniques

Appendix B. Condition Specification Syntax

B.1 Regular Expressions for Tokens

In this section, the set of tokens recognized by the lexical analyzer (Lex) are represented by regular expressions. The usual notation for regular expressions is followed. If *a* and *b* are tokens, or regular expressions, then the following tokens are also regular expressions:

ε	(the null symbol),
a | b	(alternation),
ab	(concatenation),
a*	(Kleene star - zero or more), and
(a)	(grouping).

For the benefit of the reader, each group of tokens shown below is preceded by an identifying term in bold print. Keywords are reserved words in the CS that are not case sensitive, but identifiers are case sensitive. For example, *WHEN*, *When*, and *when* are all acceptable forms of the keyword *WHEN*, but *ATTRIB*, *Attrib*, and *attrib* are treated as three separate legal identifiers. The complete set of tokens appears below.

keywords: CONDITION_SPEC | OBJECT_SPEC | TRANSITION_SPEC | FUNCTION_SPEC | REPORT_SPEC | OBJECT | AC | STATUS | TEMPORAL | TRANSITIONAL | PERMANENT | INDICATIVE | HIERARCHICAL | COORDINATE | RELATIONAL | BOOLEAN | SIGNAL | CHARACTER | STRING | INTEGER | REAL | POSITIVE | NONNEGATIVE | NONPOSITIVE | NEGATIVE | WHEN | AFTER | START | STOP | FOR | ALL | SOME | TO | DO | END | INPUT | OUTPUT | SET | CANCEL | CREATE | DESTROY

Boolean operator keywords: AND | OR | NOT

arithmetic operators: + | - | * | / | **

relational operators: < | > | <= | >= | = | <>

special symbols: (|) | [|] | := | , | : | ; | . | ..

Boolean constant keywords: TRUE | FALSE

integer constant: (+ | - | ε) (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)*

real constant: (+ | - | ε) (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)* . (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)* ((E (+ | - | ε) (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)*) | ε)

character constant: " *any one character except a quote* "

string constant: " (*any one character except a quote*)* "

identifier: (a | b | . . . | z | A | B | . . . | Z) (a | b | . . . | z | A | B | . . . | Z | 0 | 1 | . . . | 9 | _)*

comment: { (*anything except a brace*)* }

B.2 BNF Grammar

The grammar, written in Backus-Naur Form (BNF), in this section describes the language recognized by the parser component, Yacc, of the Model Analyzer. Terminals are denoted by names or symbols in bold type, and nonterminals are denoted by names in double angle symbols. The operator precedence from highest to lowest is:

******
- (unary minus)
*****, /
+, -
relop (relational operators)
NOT
AND
OR

The following is the Condition Specification written as a BNF grammar:

<<cond_spec>> ::= **CONDITION_SPEC untypedid** <<object_section>>
<<cond_section>> <<funct_section>> <<report_section>>

<<funct_section>> ::= <<null>>
| **FUNCTION_SPEC csid** <<funct_list>>

<<funct_list>> ::= <<null>>

<<report_section>> ::= <<null>>
| **REPORT_SPEC csid** <<report_list>>

<<report_list>> ::= <<null>>

<<object_section>> ::= **OBJECT_SPEC csid** <<object_list>>

<<object_list>> ::= <<obj_spec>>
| <<object_list>> <<obj_spec>>

<<obj_spec>> ::= **OBJECT** <<object_def>> <<attr_decl_list>>

<<object_def>> ::= **untypedid**
| **untypedid** [<<index_def>>]

<<index_def>> ::= <<range>> .. <<range>>
| <<index_def>> , <<range>> .. <<range>>

<<range>> ::= **untypedid**
| **intconst**
| <<variable_spec>>

<<attr_decl_list>> ::= <<attr_decl>>
| <<attr_decl_list>> <<attr_decl>>

<<attr_decl>> ::= <<attribute>> : <<cm_type>> , <<attr_type>> ;

<<attribute>> ::= **untypedid**
| **untypedid** [<<index_def>>]

<<cm_type>> ::= <<indicative_prefix>> **INDICATIVE**
| <<relational_prefix>> **RELATIONAL**

<<indicative_prefix>> ::= **STATUS TRANSITIONAL**
| **TEMPORAL TRANSITIONAL**
| **PERMANENT**

<<relational_prefix>> ::= **HIERARCHICAL**
| **COORDINATE**

<<attr_type>> ::= <<numeric_type>>
| **BOOLEAN**
| **SIGNAL**
| <<enumeration>>
| **CHARACTER**
| **STRING**

<<numeric_type>> ::= <<type_qual>> **INTEGER**
| <<type_qual>> **REAL**

<<type_qual>> ::= <<null>>
| **POSITIVE**
| **NONNEGATIVE**
| **NONPOSITIVE**
| **NEGATIVE**

<<enumeration>> ::= **(** <<enum_list>> **)**

<<enum_list>> ::= <<enum_def>>
| <<enum_list>> **,** <<enum_def>>

<<enum_def>> ::= **untypedid**
| **untypedid [** <<index_def>> **]**

<<cond_section>> ::= **TRANSITION_SPEC csid** <<ac_section>>

<<ac_section>> ::= <<action_cluster>>
| <<ac_section>> <<action_cluster>>

<<action_cluster>> ::= **AC : acid** <<condition>> **:** <<actions>> **;**

<<condition>> ::= <<time_cond>>
| <<state_cond>>

<<time_cond>> ::= <<when_exp>>
| <<after_exp>>

<<when_exp>> ::= **WHEN ALARM** (<<signal_exp>> <<parameter_section>>)

<<after_exp>> ::= **AFTER ALARM** (<<signal_exp>> **AND** <<attribute_exp>>
<<parameter_section>>)

<parameter_section>> ::= <<null>>
| , <<exp_list>>

<<signal_exp>> ::= <<signal_spec>>
| <<signal_exp>> **OR** <<signal_exp>>
| <<signal_exp>> **AND** <<signal_exp>>
| **NOT** <<signal_exp>>
| (<<signal_exp>>)

<<signal_spec>> ::= <<object_ref>> . <<indexed_sig>>

<<object_ref>> ::= **objid**
| **objid** <<index_list>>

<<indexed_sig>> ::= **signalid**
| **signalid** <<index_list>>

<<index_list>> ::= [<<exp_list>>]

<<exp_list>> ::= <<attribute_exp>>
| <<exp_list>> , <<attribute_exp>>

<<arith_exp>> ::= <<attribute_exp>> + <<attribute_exp>>
| <<attribute_exp>> - <<attribute_exp>>
| <<attribute_exp>> * <<attribute_exp>>
| <<attribute_exp>> / <<attribute_exp>>
| <<attribute_exp>> ** <<attribute_exp>>
| - <<attribute_exp>>

<<boolean_exp>> ::= <<attribute_exp>> **relop** <<attribute_exp>>
| <<attribute_exp>> **OR** <<attribute_exp>>
| <<attribute_exp>> **AND** <<attribute_exp>>
| **NOT** <<attribute_exp>>
| <<compound_exp>>

<<attribute_exp>> ::= <<arith_exp>>
| <<boolean_exp>>
| <<value_spec>>
| <<function_call>>
| (<<attribute_exp>>)

<<value_spec>> ::= **intconst**
| **realconst**
| **boolconst**
| **charconst**
| **stringconst**
| <<enum_spec>>
| <<variable_spec>>

<<enum_spec>> ::= **enumconst**
| **enumconst** <<index_list>>

<<variable_spec>> ::= <<object_ref>> **.** <<indexed_var>>
| **localvar**

<<indexed_var>> ::= <<id_name>>
| <<id_name>> <<index_list>>

<<id_name>> ::= **intid**
| **realid**
| **boolid**
| **enumid**
| **charid**
| **stringid**

<<state_cond>> ::= **WHEN** <<cond_exp>>

<<cond_exp>> ::= **START**
| <<attribute_exp>>

<<compound_cond>> ::= **FOR ALL** <<var_range>> <<attribute_exp>>
| **FOR SOME** <<var_range>> <<attribute_exp>>

<<var_range>> ::= <<variable_spec>> **:** <<attribute_exp>> **..** <<attribute_exp>> **,**

<<actions>> ::= <<action_exp>>
| <<actions>> **;** <<action_exp>>
<<action_exp>> ::= **STOP**
| <<simple_act>>
| <<compound_act>>

<<compound_act>> ::= **FOR** <<variable_spec>> **:=** <<attribute_exp>> **TO**
<<attribute_exp>> **DO** <<actions>> **; END FOR**

<<simple_act>> ::= <<assignment>>
| <<input_stmt>>
| <<output_stmt>>
| <<set_alarm>>
| <<cancel_alarm>>
| <<create_obj>>
| <<destroy_obj>>
| <<function_call>>

<<assignment>> ::= <<variable_spec>> **:=** <<attribute_exp>>

<<input_stmt>> ::= **INPUT (** <<variable_list>> **)**

<<variable_list>> **::=** <<variable_spec>>
| <<variable_list>> **,** <<variable_spec>>

<<output_stmt>> ::= **OUTPUT (** <<exp_list>> **)**

<<set_alarm>> ::= **SET ALARM (** <<signal_spec>> **,** <<attribute_exp>>
<<parameter_section>> **)**

<<cancel_alarm>> ::= **CANCEL ALARM (** <<signal_spec>> <<parameter_section>> **)**

<<create_obj>> ::= **CREATE (** <<object_list>> **)**

<<destroy_obj>> ::= **DESTROY (** <<object_list>> **)**

<<object_list>> ::= <<object_ref>>
| <<object_list>> , <<object_ref>>

<<function_call>> ::= **funvar** (<<parameter_list>>)

<<parameter_list>> ::= <<exp_list>>
| <<null>>

<<null>> ::=

Appendix C. Condition Specification Examples

C.1 Single Server Queue CS

```
{ The Condition Specification for the Single Server Queue Model }

CONDITION_SPEC single_server_queue

OBJECT_SPEC single_server_queue

   OBJECT system
      systemTime:   TEMPORAL TRANSITIONAL INDICATIVE,  NONNEGATIVE
                                                          REAL;
      stopNum:      PERMANENT INDICATIVE,              POSITIVE INTEGER;

   OBJECT server
      serverStatus:  STATUS TRANSITIONAL INDICATIVE,  (busy, idle);
      meanServiceTime:  PERMANENT INDICATIVE,         NONNEGATIVE
                                                      REAL;
      endService:    TEMPORAL TRANSITIONAL INDICATIVE, SIGNAL;
      numServed:     STATUS TRANSITIONAL INDICATIVE,  NONNEGATIVE
                                                      INTEGER;

   OBJECT parts
      numWaiting:  STATUS TRANSITIONAL INDICATIVE,  NONNEGATIVE
                                                      INTEGER;
      meanInterarrivalTime:  PERMANENT INDICATIVE,    NONNEGATIVE
                                                      REAL;
      arrival:       TEMPORAL TRANSITIONAL INDICATIVE, SIGNAL;


TRANSITION_SPEC single_server_queue

   AC: Initialization
      WHEN START:
         system.systemTime := 0;
         server.serverStatus := idle;
         parts.numWaiting := 0;
```

```
      server.numServed := 0;
      INPUT (parts.meanInterarrivalTime, server.meanServiceTime,
        system.stopNum);
      SET ALARM (parts.arrival, 0);

  AC: Termination
    WHEN server.numServed >= system.stopNum:
      OUTPUT (system.systemTime);
      STOP;

  AC: Arrival
    WHEN ALARM (parts.arrival):
      parts.numWaiting := parts.numWaiting + 1;
      SET ALARM (parts.arrival, negexp(parts.meanInterarrivalTime));

  AC: Begin_Service
    WHEN parts.numWaiting > 0 AND server.serverStatus = idle:
      parts.numWaiting := parts.numWaiting - 1;
      server.serverStatus := busy;
      SET ALARM (server.endService, negexp(server.meanServiceTime));

  AC: End_Service
    WHEN ALARM (server.endService):
      server.numServed := server.numServed + 1;
      server.serverStatus := idle;
```

C.2 First Failed, First Fixed Machine Repairman CS

```
{ The CS for the First Failed First Fixed Machine Repairman Model }

CONDITION_SPEC fourf_repairman

OBJECT_SPEC fourf_repairman

  OBJECT mrp
    system_time:  TEMPORAL TRANSITIONAL INDICATIVE,  POSITIVE REAL;
    n:            PERMANENT INDICATIVE,              POSITIVE INTEGER;
    mean_uptime: PERMANENT INDICATIVE,               POSITIVE REAL;
    mean_repairtime: PERMANENT INDICATIVE,           POSITIVE REAL;
    max_repairs:  PERMANENT INDICATIVE,              POSITIVE INTEGER;
    end_repair:   TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
```

```
    arr_fac:      TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
    arr_idle:     TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
    failure:      TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
    failq:        STATUS TRANSITIONAL INDICATIVE,    NONNEGATIVE
                                                     INTEGER;

  OBJECT facility [1..mrp.n]
    failed:       STATUS TRANSITIONAL INDICATIVE,    BOOLEAN;

  OBJECT repairman
    r_status:     STATUS TRANSITIONAL INDICATIVE,    (avail, busy, travel);
    location:     STATUS TRANSITIONAL INDICATIVE,    (fac[1..mrp.n], idle);
    num_repairs:  STATUS TRANSITIONAL INDICATIVE,    NONNEGATIVE
                                                     INTEGER;


TRANSITION_SPEC fourf_repairman

  AC: Initialization
    WHEN START:
      INPUT (mrp.n, mrp.max_repairs, mrp.mean_uptime, mrp.mean_repairtime);
      CREATE (repairman);
      FOR i := 1 TO mrp.n DO
        CREATE (facility[i]);
        facility[i].failed := FALSE;
        SET ALARM (mrp.failure, neg_exp(mrp.mean_uptime), i);
      END FOR;
      repairman.num_repairs := 0;
      repairman.location := idle;
      repairman.r_status := avail;

  AC: Termination
    WHEN repairman.num_repairs >= mrp.max_repairs:
      STOP;

  AC: Failure
    WHEN ALARM (mrp.failure, i):
      facility[i].failed := TRUE;
      Qinsert(mrp.failq, i);
```

```
AC: Begin_repair
   WHEN ALARM (mrp.arr_fac, i):
      SET ALARM (mrp.end_repair, neg_exp(mrp.mean_repairtime), i);
      repairman.r_status := busy;
      repairman.location := fac[i];

AC: End_repair
   WHEN ALARM (mrp.end_repair, i):
      SET ALARM (mrp.failure, neg_exp(mrp.mean_uptime), i);
      facility[i].failed := FALSE;
      repairman.r_status := avail;
      repairman.num_repairs := repairman.num_repairs + 1;

AC: Travel_to_idle
   WHEN (FOR ALL i:1..mrp.n, NOT facility[i].failed)  AND
   repairman.r_status = avail  AND  repairman.location <> idle:
      SET ALARM (mrp.arr_idle, traveltime(repairman.location, idle));
      repairman.r_status := travel;

AC: Arrive_idle
   WHEN ALARM (mrp.arr_idle):
      repairman.r_status := avail;
      repairman.location := idle;

AC: Travel_to_facility
   WHEN repairman.r_status = avail AND (FOR SOME i:1..mrp.n, facility[i].failed):
      SET ALARM (mrp.arr_fac, traveltime(repairman.location,
         fac[Qfirst(mrp.failq)]), i);
      Qdelete(mrp.failq);
      repairman.r_status := travel;
```

C.3 Minimum Distance Machine Repairman CS

```
{ This is a comment line.                                  }
{ The Minimal Distance Repairman Model or also called      }
{ The Closest Failed Facility Machine Repairman Problem    }

CONDITION_SPEC min_distance_repairman

OBJECT_SPEC min_distance_repairman
```

```
OBJECT mrp
   system_time:  TEMPORAL TRANSITIONAL INDICATIVE,  POSITIVE REAL;
   n:            PERMANENT INDICATIVE,              POSITIVE INTEGER;
   mean_uptime: PERMANENT INDICATIVE,               POSITIVE REAL;
   mean_repairtime:  PERMANENT INDICATIVE,          POSITIVE REAL;
   max_repairs:  PERMANENT INDICATIVE,              POSITIVE INTEGER;
   end_repair:   TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
   arr_fac:      TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
   arr_idle:     TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
   failure:      TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;

OBJECT facility [1..mrp.n]
   failed:       STATUS TRANSITIONAL INDICATIVE,    BOOLEAN;

OBJECT repairman
   r_status:     STATUS TRANSITIONAL INDICATIVE,    (avail, busy, travel);
   location:     STATUS TRANSITIONAL INDICATIVE,    (fac[1..mrp.n], idle);
   num_repairs:  STATUS TRANSITIONAL INDICATIVE,    NONNEGATIVE
                                                    INTEGER;


TRANSITION_SPEC min_distance_repairman

AC: Initialization
   WHEN START:
      INPUT (mrp.n, mrp.max_repairs, mrp.mean_uptime, mrp.mean_repairtime);
      CREATE (repairman);
      FOR i := 1 TO mrp.n DO
         CREATE (facility[i]);
         facility[i].failed := FALSE;
         SET ALARM (mrp.failure, neg_exp(mrp.mean_uptime), i);
      END FOR;
      repairman.num_repairs := 0;
      repairman.location := idle;
      repairman.r_status := avail;

AC: Termination
   WHEN repairman.num_repairs >= mrp.max_repairs:
      STOP;

AC: Failure
   WHEN ALARM (mrp.failure, i):
      facility[i].failed := TRUE;
```

```
AC: Begin_repair
   WHEN ALARM (mrp.arr_fac, i):
      SET ALARM (mrp.end_repair, neg_exp(mrp.mean_repairtime), i);
      repairman.r_status := busy;
      repairman.location := fac[i];

AC: End_repair
   WHEN ALARM (mrp.end_repair, i):
      SET ALARM (mrp.failure, neg_exp(mrp.mean_uptime), i);
      facility[i].failed := FALSE;
      repairman.r_status := avail;
      repairman.num_repairs := repairman.num_repairs + 1;

AC: Travel_to_idle
   WHEN (FOR ALL i:1..mrp.n, NOT facility[i].failed)  AND
      repairman.r_status = avail  AND  repairman.location <> idle:
      SET ALARM (mrp.arr_idle, traveltime(repairman.location, idle));
      repairman.r_status := travel;

AC: Arrive_idle
   WHEN ALARM (mrp.arr_idle):
      repairman.r_status := avail;
      repairman.location := idle;

AC: Travel_to_facility
   WHEN repairman.r_status = avail AND (FOR SOME i:1..mrp.n, facility[i].failed):
   SET ALARM (mrp.arr_fac, traveltime(repairman.location,
      closest_failed_fac(facility.failed, repairman.location)), i);
   repairman.r_status := travel;
```

C.4 Patrolling Machine Repairman CS

```
{ The CS for the Patrolling Machine Repairman problem. }

CONDITION_SPEC patrolling_repairman

OBJECT_SPEC patrolling_repairman

   OBJECT mrp
      system_time:  TEMPORAL TRANSITIONAL INDICATIVE,  POSITIVE REAL;
      n:            PERMANENT INDICATIVE,              POSITIVE INTEGER;
```

```
    mean_uptime: PERMANENT INDICATIVE,                    POSITIVE REAL;
    mean_repairtime: PERMANENT INDICATIVE,                POSITIVE REAL;
    traveltime:   PERMANENT INDICATIVE,                   POSITIVE REAL;
    max_repairs:  PERMANENT INDICATIVE,                   POSITIVE INTEGER;
    end_repair:   TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
    arrive:       TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
    failure:      TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;

  OBJECT facility [1..mrp.n]
    failed:       STATUS TRANSITIONAL INDICATIVE,  BOOLEAN;

  OBJECT repairman
    location:     STATUS TRANSITIONAL INDICATIVE,  (fac[1..mrp.n], idle);
    num_repairs:  STATUS TRANSITIONAL INDICATIVE,  NONNEGATIVE
                                                   INTEGER;


TRANSITION_SPEC patrolling_repairman

  AC: Initialization
    WHEN START:
      INPUT (mrp.n, mrp.max_repairs, mrp.mean_uptime, mrp.mean_repairtime,
        mrp.traveltime);
      CREATE (repairman);
      FOR i := 1 TO mrp.n DO
        CREATE (facility[i]);
        facility[i].failed := FALSE;
        SET ALARM (mrp.failure, neg_exp(mrp.mean_uptime), i);
      END FOR;
      repairman.num_repairs := 0;
      repairman.location := fac[1];

  AC: Termination
    WHEN repairman.num_repairs >= mrp.max_repairs:
      STOP;

  AC: Failure
    WHEN ALARM (mrp.failure, i):
      facility[i].failed := TRUE;

  AC: Begin_repair
    WHEN repairman.location = fac[i]  AND  facility[i].failed:
      SET ALARM (mrp.end_repair, neg_exp(mrp.mean_repairtime), i);
```

```
AC: End_of_repair
   WHEN ALARM (mrp.end_repair, i):
      SET ALARM (mrp.failure, neg_exp(mrp.mean_uptime), i);
      facility[i].failed := FALSE;
      repairman.num_repairs := repairman.num_repairs + 1;

AC: Travel_to_facility
   WHEN repairman.location = fac[i]  AND  (NOT facility[i].failed):
      SET ALARM (mrp.arrive, neg_exp(mrp.traveltime), i);

AC: Arrive_at_facility
   WHEN ALARM (mrp.arrive, i):
      i := MOD(i + 1, mrp.n);
      repairman.location := fac[i];
```

C.5 Harbor Model CS

```
{ The Condition Specification for the Harbor Model }

CONDITION_SPEC harbor

OBJECT_SPEC harbor

   OBJECT environment
      system_time:  TEMPORAL TRANSITIONAL INDICATIVE,  NONNEGATIVE
                                                          REAL;
      m1:            PERMANENT INDICATIVE,             POSITIVE REAL;
      m2:           PERMANENT INDICATIVE,              POSITIVE REAL;
      s:            PERMANENT INDICATIVE,              POSITIVE REAL;
      k1:           PERMANENT INDICATIVE,              POSITIVE REAL;
      k2:           PERMANENT INDICATIVE,              POSITIVE REAL;
      num_berths:   PERMANENT INDICATIVE,              POSITIVE INTEGER;
      num_tugs:     PERMANENT INDICATIVE,              POSITIVE INTEGER;

   OBJECT arrival_area
      arrival:      TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
      num_arr_q:    HIERARCHICAL RELATIONAL,    NONNEGATIVE INTEGER;
      exit_count:   COORDINATE RELATIONAL,      NONNEGATIVE INTEGER;
```

```
OBJECT pier
   num_free_berths: STATUS TRANSITIONAL INDICATIVE, NONNEGATIVE
                                                        INTEGER;
   num_depart_q: STATUS TRANSITIONAL INDICATIVE, NONNEGATIVE
                                                        INTEGER;
   end_unload:   TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;

OBJECT tugs
   num_pier_tugs: STATUS TRANSITIONAL INDICATIVE,NONNEGATIVE
                                                        INTEGER;
   num_arr_tugs: STATUS TRANSITIONAL INDICATIVE,  NONNEGATIVE
                                                        INTEGER;
   end_enter:    TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
   end_deberth:  TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
   tug_at_pier:  TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
   tug_at_ocean: TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;


TRANSITION_SPEC harbor

AC: Initialization
   WHEN START:
      CREATE (arrival_area, pier, tugs);
      environment.num_tugs := 2;
      environment.num_berths := 8;
      tugs.num_pier_tugs := environment.num_tugs;
      tugs.num_arr_tugs := 0;
      pier.num_free_berths := environment.num_berths;
      pier.num_depart_q := 0;
      arrival_area.num_arr_q := 0;
      arrival_area.exit_count := 0;
      environment.system_time := 0;
      SET ALARM (arrival_area.arrival, 0);

AC: Termination
   WHEN arrival_area.exit_count >= 100:
      STOP;

AC: Arrival
   WHEN ALARM (arrival_area.arrival):
      SET ALARM (arrival_area.arrival, arrival_time(environment.m1));
      arrival_area.num_arr_q := arrival_area.num_arr_q + 1;
```

```
AC: Enter
   WHEN arrival_area.num_arr_q > 0  AND  tugs.num_arr_tugs > 0  AND
   pier.num_free_berths > 0:
      SET ALARM (tugs.end_enter, enter_time(environment.k1));
      arrival_area.num_arr_q := arrival_area.num_arr_q - 1;
      tugs.num_arr_tugs := tugs.num_arr_tugs - 1;
      pier.num_free_berths := pier.num_free_berths - 1;

AC: Unload
   WHEN ALARM (tugs.end_enter):
      SET ALARM (pier.end_unload, unload_time(environment.m2,environment.s));
      tugs.num_pier_tugs := tugs.num_pier_tugs + 1;

AC: End_unload
   WHEN ALARM (pier.end_unload):
      pier.num_depart_q := pier.num_depart_q + 1;

AC: Deberth
   WHEN pier.num_depart_q > 0  AND  tugs.num_pier_tugs > 0  AND
   (pier.num_free_berths = 0  OR  arrival_area.num_arr_q = 0):
      SET ALARM (tugs.end_deberth, deberth_time(environment.k1));
      tugs.num_pier_tugs := tugs.num_pier_tugs - 1;
      pier.num_depart_q := pier.num_depart_q - 1;
      pier.num_free_berths := pier.num_free_berths + 1;

AC: End_deberth
   WHEN ALARM (tugs.end_deberth):
      arrival_area.exit_count := arrival_area.exit_count + 1;
      tugs.num_arr_tugs := tugs.num_arr_tugs + 1;

AC: Move_tug_to_pier
   WHEN pier.num_depart_q > 0  AND  tugs.num_pier_tugs = 0  AND
   tugs.num_arr_tugs > 0  AND
   (arrival_area.num_arr_q = 0  OR  pier.num_free_berths = 0):
      SET ALARM (tugs.tug_at_pier, move_tug_time(environment.k2));
      tugs.num_arr_tugs := tugs.num_arr_tugs - 1;

AC: Tug_arr_at_pier
   WHEN ALARM (tugs.tug_at_pier):
      tugs.num_pier_tugs := tugs.num_pier_tugs + 1;
```

```
AC: Move_tug_to_ocean
   WHEN arrival_area.num_arr_q > 0  AND  tugs.num_arr_tugs = 0  AND
   tugs.num_pier_tugs > 0  AND  pier.num_free_berths > 0:
      SET ALARM (tugs.tug_at_ocean, move_tug_time(environment.k2));
      tugs.num_pier_tugs := tugs.num_pier_tugs - 1;

AC: Tug_arr_at_ocean
   WHEN ALARM (tugs.tug_at_ocean):
      tugs.num_arr_tugs := tugs.num_arr_tugs + 1;
```

C.6 Automatic Inspection and Repair Unit CS

```
{ The Condition Specification for the Automatic Inspection and Repair Unit  }
{ polling problem model.                                                    }

CONDITION_SPEC AIRU_model

OBJECT_SPEC AIRU_model

   OBJECT environment
      time:        TEMPORAL TRANSITIONAL INDICATIVE,  POSITIVE REAL;
      done_tm:     TEMPORAL TRANSITIONAL INDICATIVE,  POSITIVE REAL;
      wrm_tm:      TEMPORAL TRANSITIONAL INDICATIVE,  POSITIVE REAL;
      done:        TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
      warm:        TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
      m_failure:   PERMANENT INDICATIVE,              POSITIVE REAL;
      m_r_tm:      PERMANENT INDICATIVE,              POSITIVE REAL;
      bigT:        PERMANENT INDICATIVE,              POSITIVE INTEGER;
      n:           PERMANENT INDICATIVE,              POSITIVE INTEGER;
      trv_tm:      PERMANENT INDICATIVE,              POSITIVE INTEGER;
      sum_d_tm:    TEMPORAL TRANSITIONAL INDICATIVE,  POSITIVE REAL;
      percent_up:  PERMANENT INDICATIVE,              POSITIVE REAL;
      temp:        PERMANENT INDICATIVE,              POSITIVE REAL;
      efficiency:  PERMANENT INDICATIVE,              POSITIVE REAL;

   OBJECT sites[1..environment.n]
      failed:      STATUS TRANSITIONAL INDICATIVE,    BOOLEAN;
      d_cnt:       STATUS TRANSITIONAL INDICATIVE,    POSITIVE INTEGER;
      begin_d:     TEMPORAL TRANSITIONAL INDICATIVE,  SIGNAL;
      td_tm:       TEMPORAL TRANSITIONAL INDICATIVE,  POSITIVE REAL;
      nxt_d_tm:    TEMPORAL TRANSITIONAL INDICATIVE,  POSITIVE REAL;
```

```
OBJECT airu
   r_cnt:      STATUS TRANSITIONAL INDICATIVE,    POSITIVE INTEGER;
   a_status:   STATUS TRANSITIONAL INDICATIVE,    (idle, busy);
   loc:        STATUS TRANSITIONAL INDICATIVE,    POSITIVE INTEGER;
   nxt_loc:    STATUS TRANSITIONAL INDICATIVE,    POSITIVE INTEGER;
   r_tm:       TEMPORAL TRANSITIONAL INDICATIVE, POSITIVE REAL;
   arr_tm:     TEMPORAL TRANSITIONAL INDICATIVE, SIGNAL;
   begin_r:    STATUS TRANSITIONAL INDICATIVE,    BOOLEAN;
   end_r:      TEMPORAL TRANSITIONAL INDICATIVE, SIGNAL;
   t_r_tm:     TEMPORAL TRANSITIONAL INDICATIVE, POSITIVE REAL;


TRANSITION_SPEC AIRU_model

   AC: Initialization
      WHEN START:
         INPUT (environment.n, environment.m_failure, environment.m_r_tm,
            environment.bigT, environment.done_tm, environment.wrm_tm);
         FOR i := 1 TO environment.n DO
            CREATE (sites[i]);
            sites[i].failed := FALSE;
            sites[i].d_cnt := 0;
            sites[i].td_tm := 0.0;
            SET ALARM (sites[i].begin_d, POISSON(environment.m_failure));
         END FOR;
         CREATE (airu);
         airu.r_cnt := 0;
         airu.t_r_tm := 0.0;
         airu.a_status := idle;
         airu.begin_r := FALSE;
         environment.trv_tm:= environment.bigT + 1;
         SET ALARM (airu.arr_tm, environment.time);
         SET ALARM (environment.done, environment.done_tm);
         SET ALARM (environment.warm, environment.wrm_tm);

   AC: Fails
      WHEN ALARM (sites[i].begin_d):
         sites[i].failed := TRUE;
         sites[i].d_cnt := sites[i].d_cnt + 1;
```

```
AC: Arrive
   WHEN ALARM (airu.arr_tm):
      airu.loc := airu.nxt_loc;
      airu.nxt_loc := MOD(airu.loc, environment.n) + 1;
      airu.a_status := idle;

AC: Broke
   WHEN airu.a_status = idle AND sites[airu.loc].failed:
      airu.a_status := busy;
      airu.begin_r := TRUE;

AC: Move
   WHEN airu.a_status = idle AND NOT sites[airu.loc].failed:
      airu.a_status := busy;
      SET ALARM (airu.arr_tm, environment.time + environment.trv_tm);

AC: Repair
   WHEN airu.begin_r:
      airu.begin_r := FALSE;
      airu.r_tm := NEG_EXP(environment.m_r_tm);
      airu.t_r_tm := airu.t_r_tm + airu.r_tm;
      SET ALARM (airu.end_r, environment.time + airu.r_tm);

AC: Endrep
   WHEN ALARM (airu.end_r):
      sites[airu.loc].failed := FALSE;
      sites[airu.loc].td_tm := sites[airu.loc].td_tm +
         (environment.time - sites[airu.loc].nxt_d_tm);
      airu.r_cnt := airu.r_cnt + 1;
      sites[airu.loc].nxt_d_tm := environment.time +
         POISSON(environment.m_failure);
      SET ALARM (sites[airu.loc].begin_d, sites[airu.loc].nxt_d_tm);
      SET ALARM (airu.arr_tm, environment.time + environment.trv_tm);

AC: Reset
   WHEN ALARM (environment.warm):
      FOR i := 1 TO environment.n DO
         sites[i].d_cnt := 0;
         sites[i].td_tm := 0.0;
      END FOR;
      airu.r_cnt := 0;
      airu.t_r_tm := 0.0;
```

```
AC: Termination
  WHEN ALARM (environment.done):
    environment.temp := environment.done_tm - environment.wrm_tm;
    environment.sum_d_tm := 0.0;
    FOR i := 1 TO environment.n DO
      environment.sum_d_tm := environment.sum_d_tm + sites[i].td_tm;
      environment.percent_up := (1.0 - (sites[i].td_tm /
        (environment.n * environment.temp))) * 100.0;
      OUTPUT ("SITE NUMBER ", i);
      OUTPUT ("DOWN COUNT = ", sites[i].d_cnt);
      OUTPUT ("DOWN TIME = ", sites[i].td_tm);
      OUTPUT ("PERCENT UP = ", environment.percent_up);
    END FOR;
    OUTPUT ("REPAIR COUNT = ", airu.r_cnt);
    OUTPUT ("REPAIR TIME = ", airu.t_r_tm);
    environment.efficiency := (1.0 - (environment.sum_d_tm /
      (environment.n * environment.temp))) * 100.0;
    OUTPUT ("EFFICIENCY = ", environment.efficiency);
    STOP;
```

Appendix D. Expert System Condition Specifications

D.1 Single Server Queue CS

```
CONDITION_SPEC ssqueue

OBJECT_SPEC ssqueue

   OBJECT system
      systemTime:              NONNEGATIVE REAL;
      stopNum:                 POSITIVE INTEGER;

   OBJECT server
      serverStatus:            {busy, idle};
      meanServiceTime:         NONNEGATIVE INTEGER;
      endService:              SIGNAL;
      numServed:               NONNEGATIVE INTEGER;

   OBJECT parts
      numWaiting:              NONNEGATIVE INTEGER;
      meanInterarrivalTime:    NONNEGATIVE REAL;
      arrival:                 SIGNAL;


TRANSITION_SPEC ssqueue

   Initialization:
      WHEN START
         systemTime = 0;
         serverStatus = idle;
         numWaiting = 0;
         numServed = 0;
         INPUT (meanInterarrivalTime, meanServiceTime, stopNum);
         SET ALARM (arrival, 0);
```

```
Termination:
   WHEN numServed >= stopNum
      OUTPUT (systemTime);
      STOP;

Arrival:
   WHEN ALARM (arrival)
      numWaiting = numWaiting + 1;
      SET ALARM (arrival, negexp(meanInterarrivalTime));

Begin_Service:
   WHEN numWaiting > 0  AND  serverStatus == idle
      numWaiting = numWaiting - 1;
      serverStatus = busy;
      SET ALARM (endService, negexp(meanServiceTime));

End_Service:
   WHEN ALARM (endService)
      numServed = numServed + 1;
      serverStatus = idle;
```

D.2 First Failed, First Fixed Machine Repairman CS

```
CONDITION_SPEC fourf

OBJECT_SPEC fourf

   OBJECT environment
      systemtime:          REAL;
      n:                   POSITIVE INTEGER;
      meanuptime:          REAL;
      meanrepairtime:      REAL;
      maxrepairs:          POSITIVE INTEGER;

   OBJECT facilities
      fac(i):              INTEGER;
      failure(i):          SIGNAL;
      failed(i):           BOOLEAN;
      endrepair(i):        SIGNAL;
      arrfac(i):           SIGNAL;
```

```
    numrepairs:              NONNEGATIVE INTEGER;
    failq:                   INTEGER;

  OBJECT repairman
    status:                  {available, busy, travel};
    location:                {faci, idle, travel};

  OBJECT idleposn
    arridle:                 SIGNAL;


TRANSITION_SPEC fourf

  Initialization:
    WHEN START
      INPUT (n, maxrepairs, meanuptime, meanrepairtime);
      CREATE (repairman);
      FORALL(i), CREATE (facilities(i));
      FORALL(i), failed(i) = FALSE;
      FORALL(i), SET ALARM (failure(i), negexp(meanuptime));
      numrepairs = 0;
      location = idle;
      status = available;

  Termination:
    WHEN numrepairs >= maxrepairs
      STOP;

  Failure:
    WHEN ALARM (failure(i))
      failed(i) = TRUE;
      Qinsert (failq, i);

  Begin_repair:
    WHEN ALARM (arrfac(i))
      SET ALARM (endrepair(i), negexp(meanrepairtime));
      status = busy;
      location = faci;

  End_repair:
    WHEN ALARM (endrepair(i))
      SET ALARM (failure(i), negexp(meanuptime));
      failed(i) = FALSE;
```

```
    status = available;
    numrepairs = numrepairs + 1;

  Travel_to_idle:
    WHEN FORALL(i), failed(i) AND status == available AND location <> idle
      SET ALARM (arridle, traveltime(location,idle));
      status = travel;

  Arrive_idle:
    WHEN ALARM (arridle)
      status = available;
      location = idle;

  Travel_to_facility:
    WHEN status == available AND FORSOME(i), failed(i)
      i = Qfirst (failq);
      Qdelete (failq);
      SET ALARM (arrfac(i), traveltime(location, faci));
      status = travel;
```

D.3 Harbor Model CS

```
CONDITION_SPEC harbor

OBJECT_SPEC harbor

  OBJECT environment
    system_time:        NONNEGATIVE REAL;
    m1:                 POSITIVE REAL;
    m2:                 POSITIVE REAL;
    s:                  POSITIVE REAL;
    k1:                 POSITIVE REAL;
    k2:                 POSITIVE REAL;
    num_berths:         POSITIVE INTEGER;
    num_tugs:           POSITIVE INTEGER;

  OBJECT arrival_area
    arrival:            SIGNAL;
    num_arr_q:          NONNEGATIVE INTEGER;
    exit_count:         NONNEGATIVE INTEGER;
```

```
  OBJECT pier
    num_free_berths:        NONNEGATIVE INTEGER;
    num_depart_q:           NONNEGATIVE INTEGER;
    end_unload:             SIGNAL;

  OBJECT tugs
    num_pier_tugs:          NONNEGATIVE INTEGER;
    num_arr_tugs:           NONNEGATIVE INTEGER;
    end_enter:              SIGNAL;
    end_deberth:            SIGNAL;
    tug_at_pier:            SIGNAL;
    tug_at_ocean:           SIGNAL;


TRANSITION_SPEC harbor

  Initialization:
    WHEN START
      CREATE (arrival_area);
      CREATE (pier);
      CREATE (tugs);
      num_tugs = 2;
      num_berths = 8;
      num_pier_tugs = num_tugs;
      num_arr_tugs = 0;
      num_free_berths = num_berths;
      num_depart_q = 0;
      num_arr_q = 0;
      exit_count = 0;
      system_time = 0;
      SET ALARM (arrival, 0);

  Termination:
    WHEN exit_count >= 100
      STOP;

  Arrival:
    WHEN ALARM (arrival)
      SET ALARM (arrival, arrival_time(m1));
      num_arr_q = num_arr_q + 1;
```

```
Enter:
   WHEN num_arr_q > 0  AND  num_arr_tugs > 0  AND  num_free_berths > 0
      SET ALARM (end_enter, enter_time(k1));
      num_arr_q = num_arr_q -1;
      num_arr_tugs = num_arr_tugs - 1;
      num_free_berths = num_free_berths - 1;

Unload:
   WHEN ALARM (end_enter)
      SET ALARM (end_unload, unload_time(m2,s));
      num_pier_tugs = num_pier_tugs + 1;

End_unload:
   WHEN ALARM (end_unload)
      num_depart_q = num_depart_q + 1;

Deberth:
   WHEN num_depart_q > 0 AND num_pier_tugs > 0 AND
   (num_free_berths == 0 OR num_arr_q == 0)
      SET ALARM (end_deberth, deberth_time(k1));
      num_pier_tugs = num_pier_tugs - 1;
      num_depart_q = num_depart_q - 1;
      num_free_berths = num_free_berths + 1;

End_deberth:
   WHEN ALARM (end_deberth)
      exit_count = exit_count + 1;
      num_arr_tugs = num_arr_tugs + 1;

Move_tug_to_pier:
   WHEN num_depart_q > 0 AND num_pier_tugs == 0 AND num_arr_tugs > 0 AND
   (num_arr_q == 0 OR num_free_berths == 0)
      SET ALARM (tug_at_pier, move_tug_time(k2));
      num_arr_tugs = num_arr_tugs - 1;

Tug_arr_at_pier:
   WHEN ALARM (tug_at_pier)
      num_pier_tugs = num_pier_tugs + 1;
```

```
Move_tug_to_ocean:
   WHEN num_arr_q > 0 AND num_arr_tugs == 0 AND num_pier_tugs > 0 AND
   num_free_births > 0
      SET ALARM (tug_at_ocean, move_tug_time(k2));
      num_pier_tugs = num_pier_tugs - 1;

Tug_arr_at_ocean:
   WHEN ALARM (tug_at_ocean)
      num_arr_tugs = num_arr_tugs + 1;
```

Appendix E. Prolog List Structure of Condition Specifications

E.1 Prolog List Structure of Single Server Queue CS

```
object_spec([

   [system,
      [systemTime,              'NONNEGATIVE', 'REAL'],
      [stopNum,                 'POSITIVE', 'INTEGER']],

   [server,
      [serverStatus,            [busy, idle]],
      [meanServiceTime,         'NONNEGATIVE', 'REAL'],
      [endService,              'SIGNAL'],
      [numServed,               'NONNEGATIVE', 'INTEGER']],

   [parts,
      [numWaiting,              'NONNEGATIVE', 'INTEGER'],
      [meanInterarrivalTime,    'NONNEGATIVE', 'REAL'],
      [arrival,                 'SIGNAL']]]).


transition_spec([

   ['Initialization',
      [['START']],
         [systemTime, '=', 0],
         [serverStatus, '=', idle],
         [numWaiting, '=', 0],
         [numServed, '=', 0],
         ['INPUT', '(', meanInterarrivalTime, meanServiceTime, stopNum, ')'],
         ['SET_ALARM', '(', arrival, 0, ')']],
```

```
['Termination',
   [[numServed, '>=', stopNum]],
      ['OUTPUT', '(', systemTime, ')'],
      ['STOP']],

['Arrival',
   [['ALARM', arrival]],
      [numWaiting, '=', numWaiting, '+', 1],
      ['SET_ALARM', '(', arrival, 'negexp(meanInterarrivalTime)', ')']],

['Begin_Service',
   [[numWaiting, '>', 0], [serverStatus, '==', idle]],
      [numWaiting, '=', numWaiting, '-', 1],
      [serverStatus, '=', busy],
      ['SET_ALARM', '(', endService, 'negexp(meanServiceTime)', ')']],

['End_Service',
   [['ALARM', endService]],
      [numServed, '=', numServed, '+', 1],
      [serverStatus, '=', idle]]]).
```

E.2 Prolog List Structure of First Failed, First Fixed Machine Repairman CS

```
object_spec([

   [environment,
      [systemtime,          'REAL'],
      [n,                   'POSITIVE', 'INTEGER'],
      [meanuptime,          'REAL'],
      [meanrepairtime,      'REAL'],
      [maxrepairs,          'POSITIVE','INTEGER']],

   [facilities,
      ['fac(i)',            'INTEGER'],
      ['failure(i)',        'SIGNAL'],
      ['failed(i)',         'BOOLEAN'],
      ['endrepair(i)',      'SIGNAL'],
      ['arrfac(i)',         'SIGNAL'],
      [numrepairs,          'NONNEGATIVE', 'INTEGER'],
      [failq,               'INTEGER']],
```

```
  [repairman,
     [status,                [available, busy, travel]],
     [location,              [faci, idle, travel]]],

  [idleposn,
     [arridle,               'SIGNAL']]]).


transition_spec([

  ['Initialization',
     [['START']],
        ['INPUT', '(', n, maxrepairs, meanuptime, meanrepairtime, ')'],
        ['CREATE', '(', repairman, ')'],
        ['FORALL(i)', 'CREATE', '(', 'facilities(i)', ')'],
        ['FORALL(i)', 'failed(i)', '=', 'FALSE'],
        ['FORALL(i)', 'SET_ALARM', '(', 'failure(i)', 'negexp(meanuptime)', ')'],
        [numrepairs, '=', 0],
        [location, '=', idle],
        [status, '=', available]],

  ['Termination',
     [[numrepairs, '>=', maxrepairs]],
        ['STOP']],

  ['Failure',
     [['ALARM', 'failure(i)']],
        ['failed(i)', '=', 'TRUE'],
        ['Qinsert', '(', failq, '%i', ')']],

  ['Begin_repair',
     [['ALARM', 'arrfac(i)']],
        ['SET_ALARM', '(', 'endrepair(i)', 'negexp(meanrepairtime)', ')'],
        [status, '=', busy],
        [location, '=', faci]],

  ['End_repair',
     [['ALARM', 'endrepair(i)']],
        ['SET_ALARM', '(', 'failure(i)', 'negexp(meanuptime)', ')'],
        ['failed(i)', '=', 'FALSE'],
        [status, '=', available],
        [numrepairs, '=', numrepairs, '+', 1]],
```

```
['Travel_to_idle',
   [['FORALL(i)', 'NOT', 'failed(i)'], [status, '==', available], [location, '<>', idle]],
      ['SET_ALARM', '(', arridle, 'traveltime(location,idle)', ')'],
      [status, '=', travel]],

['Arrive_idle',
   [['ALARM', arridle]],
      [status, '=', available],
      [location, '=', idle]],

['Travel_to_facility',
   [[status, '==', available], ['FORSOME(i)', 'failed(i)']],
      ['%i', '=', 'Qfirst', '(', failq, ')'],
      ['Qdelete', '(', failq, ')'],
      ['SET_ALARM', '(', 'arrfac(i)', 'traveltime(location,faci)', ')'],
      [status, '=', travel]]]).
```

E.3 Prolog List Structure of Harbor Model CS

```
object_spec([

   [environment,
      [system_time,          'NONNEGATIVE', 'REAL'],
      [m1,                   'POSITIVE', 'REAL'],
      [m2,                   'POSITIVE', 'REAL'],
      [s,                    'POSITIVE', 'REAL'],
      [k1,                   'POSITIVE', 'REAL'],
      [k2,                   'POSITIVE', 'REAL'],
      [num_berths,           'POSITIVE', 'INTEGER'],
      [num_tugs,             'POSITIVE', 'INTEGER']],

   [arrival_area,
      [arrival,              'SIGNAL'],
      [num_arr_q,            'NONNEGATIVE', 'INTEGER'],
      [exit_count,           'NONNEGATIVE', 'INTEGER']],

   [pier,
      [num_free_berths,      'NONNEGATIVE', 'INTEGER'],
      [num_depart_q,         'NONNEGATIVE', 'INTEGER'],
      [end_unload,           'SIGNAL']],
```

```
   [tugs,
      [num_pier_tugs,          'NONNEGATIVE', 'INTEGER'],
      [num_arr_tugs,           'NONNEGATIVE', 'INTEGER'],
      [end_enter,              'SIGNAL'],
      [end_deberth,            'SIGNAL'],
      [tug_at_pier,            'SIGNAL'],
      [tug_at_ocean,           'SIGNAL']]]).


transition_spec([

   ['Initialization',
      [['START']],
         ['CREATE', '(', arrival_area, ')'],
         ['CREATE', '(', pier, ')'],
         ['CREATE', '(', tugs, ')'],
         [num_tugs, '=', 2],
         [num_berths, '=', 8],
         [num_pier_tugs, '=', num_tugs],
         [num_arr_tugs, '=', 0],
         [num_free_berths, '=', num_berths],
         [num_depart_q, '=', 0],
         [num_arr_q, '=', 0],
         [exit_count, '=', 0],
         [system_time, '=', 0],
         ['SET_ALARM', '(', arrival, 0, ')']],

   ['Termination',
      [[exit_count, '>=', 100]],
         ['STOP']],

   ['Arrival',
      [['ALARM', arrival]],
         ['SET_ALARM', '(', arrival, 'arrival_time(m1)', ')'],
         [num_arr_q, '=', num_arr_q, '+', 1]],

   ['Enter',
      [[num_arr_q, '>', 0], [num_arr_tugs, '>', 0], [num_free_berths, '>', 0]],
         ['SET_ALARM', '(', end_enter, 'enter_time(k1)', ')'],
         [num_arr_q, '=', num_arr_q, '-', 1],
         [num_arr_tugs, '=', num_arr_tugs, '-', 1],
         [num_free_berths, '=', num_free_berths, '-', 1]],
```

```
['Unload',
   [['ALARM', end_enter]],
      ['SET_ALARM', '(', end_unload, 'unload_time(m2,s)', ')'],
      [num_pier_tugs, '=', num_pier_tugs, '+', 1]],

['End_unload',
   [['ALARM', end_unload]],
      [num_depart_q, '=', num_depart_q, '+', 1]],

['Deberth',
   [[num_depart_q, '>', 0], [num_pier_tugs, '>', 0],
    [[num_free_berths, '==', 0], [num_arr_q, '==', 0]]],
      ['SET_ALARM', '(', end_deberth, 'deberth_time(k1)', ')'],
      [num_pier_tugs, '=', num_pier_tugs, '-', 1],
      [num_depart_q, '=', num_depart_q, '-', 1],
      [num_free_berths, '=', num_free_berths, '+', 1]],

 ['End_deberth',
   [['ALARM', end_deberth]],
     [exit_count, '=', exit_count, '+', 1],
     [num_arr_tugs, '=', num_arr_tugs, '+', 1]],

['Move_tug_to_pier',
   [[num_depart_q, '>', 0], [num_pier_tugs, '==', 0], [num_arr_tugs, '>', 0],
    [[num_arr_q, '==', 0], [num_free_berths, '==', 0]]],
      ['SET_ALARM', '(', tug_at_pier, 'move_tug_time(k2)', ')'],
      [num_arr_tugs, '=', num_arr_tugs, '-', 1]],

['Tug_arr_at_pier',
   [['ALARM', tug_at_pier]],
      [num_pier_tugs, '=', num_pier_tugs, '+', 1]],

['Move_tug_to_ocean',
   [[num_arr_q, '>', 0], [num_arr_tugs, '==', 0], [num_pier_tugs, '>', 0],
    [num_free_berths, '>', 0]],
      ['SET_ALARM', '(', tug_at_ocean, 'move_tug_time(k2)', ')'],
       [num_pier_tugs, '=', num_pier_tugs, '-', 1]],

['Tug_arr_at_ocean',
   [['ALARM', tug_at_ocean]],
      [num_arr_tugs, '=', num_arr_tugs, '+', 1]]]).
```

Appendix F. Simplified Action Cluster Incidence Graph Results

F.1 Single Server Queue Simplified ACIG Results

Initialization --> Arrival

Arrival --> Arrival

Arrival --> Begin_Service

Begin_Service --> End_Service

End_Service --> Termination

End_Service --> Begin_Service

F.2 First Failed, First Fixed Machine Repairman Simplified ACIG Results

Initialization --> Failure

Failure --> Travel_to_facility

Begin_repair --> End_repair

End_repair --> Termination

End_repair --> Failure

End_repair --> Travel_to_idle

End_repair --> Travel_to_facility

Travel_to_idle --> Arrive_idle

Arrive_idle --> Travel_to_facility

Travel_to_facility --> Begin_repair

F.3 Harbor Model Simplified ACIG Results

Initialization --> Arrival

Arrival --> Arrival

Arrival --> Enter

Arrival --> Move_tug_to_ocean

Enter --> Unload

Enter --> Deberth

Enter --> Move_tug_to_pier

Enter --> Move_tug_to_ocean

Unload --> End_unload

Unload --> Deberth

Unload --> Move_tug_to_ocean

End_unload --> Deberth

End_unload --> Move_tug_to_pier

Deberth --> Enter

Deberth --> End_deberth

Deberth --> Move_tug_to_pier

Deberth --> Move_tug_to_ocean

End_deberth --> Termination

End_deberth --> Enter

End_deberth --> Move_tug_to_pier

Move_tug_to_pier --> Tug_arr_at_pier

Tug_arr_at_pier --> Deberth

Tug_arr_at_pier --> Move_tug_to_ocean

Move_tug_to_ocean --> Tug_arr_at_ocean

Tug_arr_at_ocean --> Enter

Tug_arr_at_ocean --> Move_tug_to_pier

Appendix G. Action Cluster Incidence Graphs

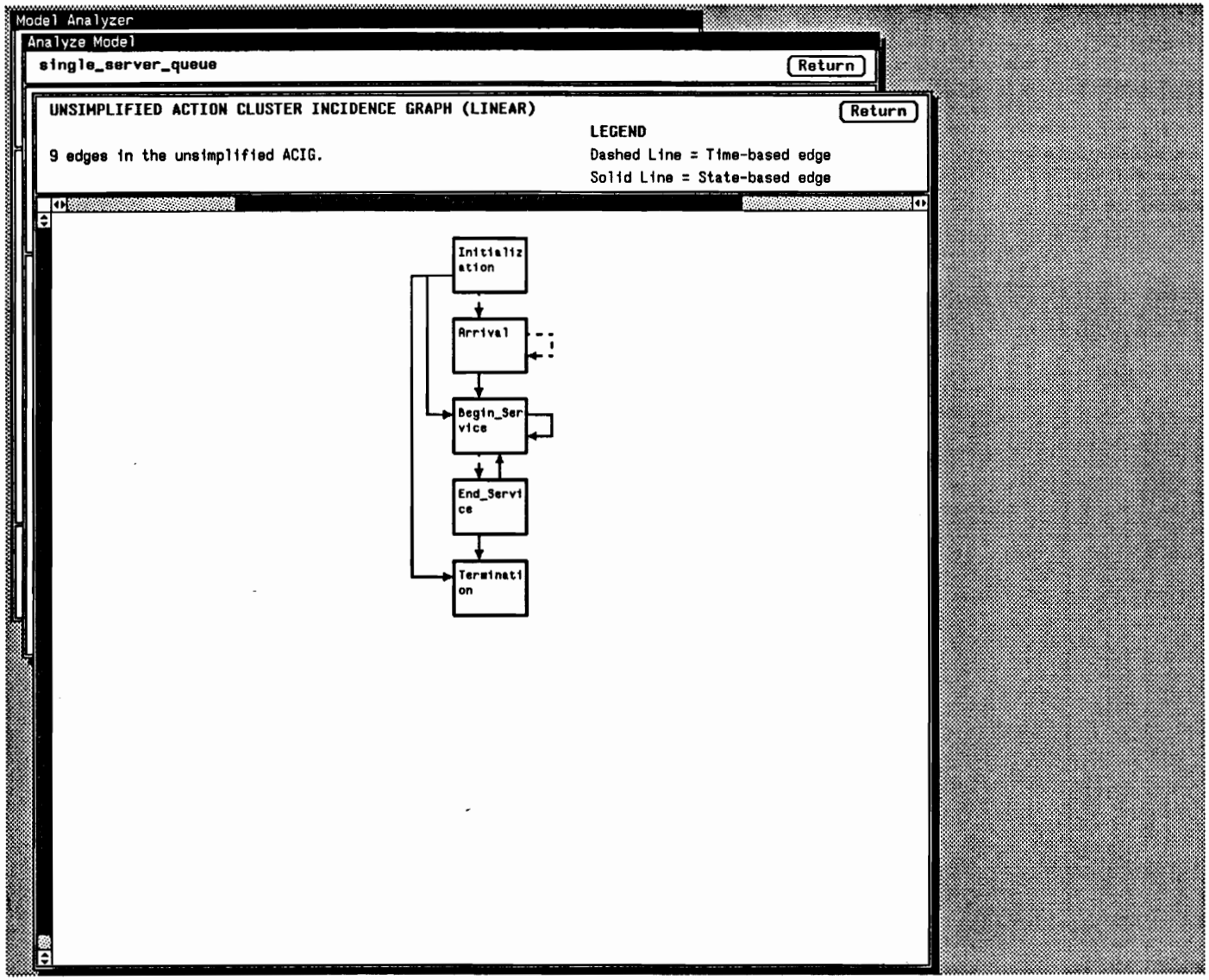


Figure 125. Unsimplified Single Server Queue Action Cluster Incidence Graph

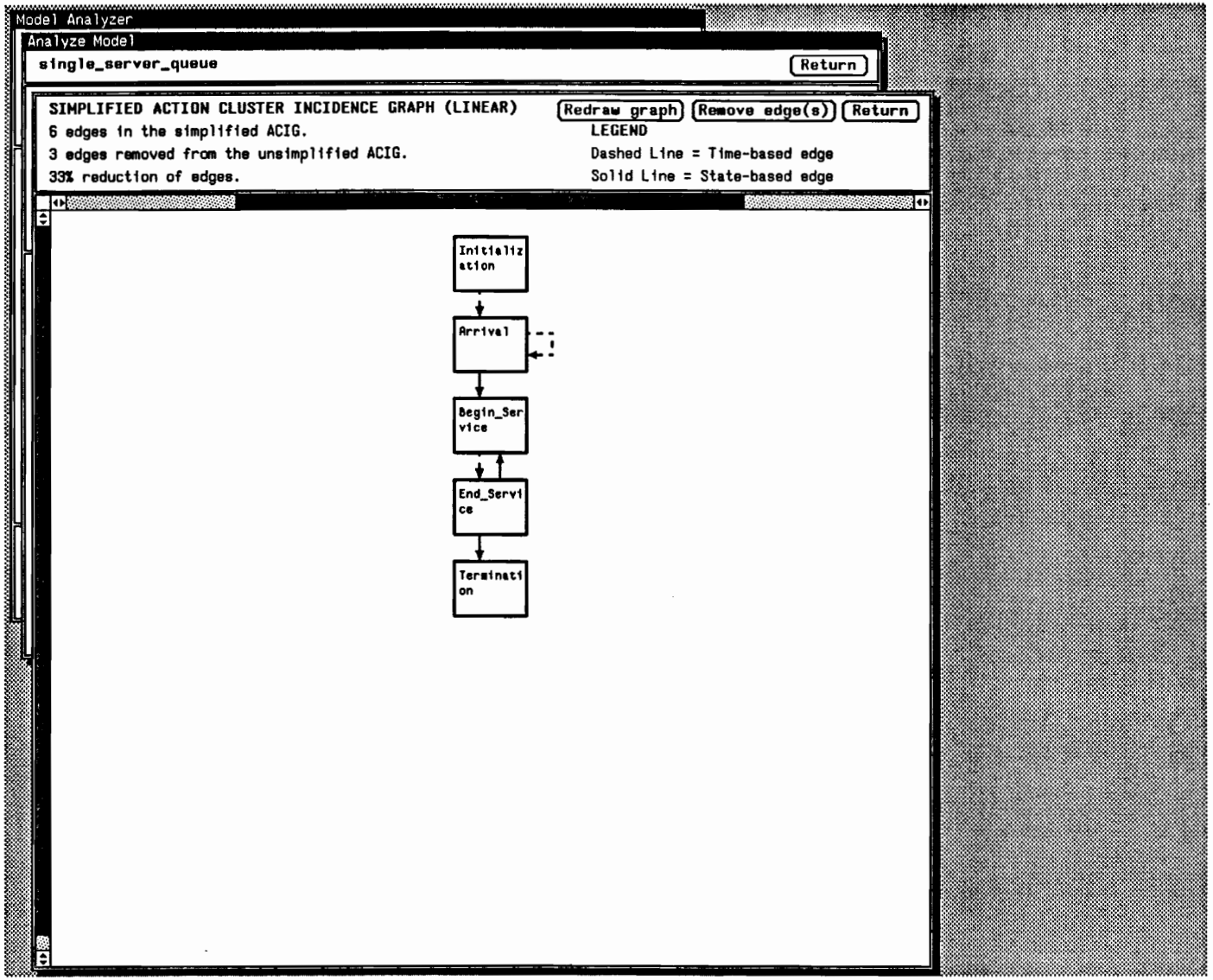


Figure 126. Simplified Single Server Queue Action Cluster Incidence Graph

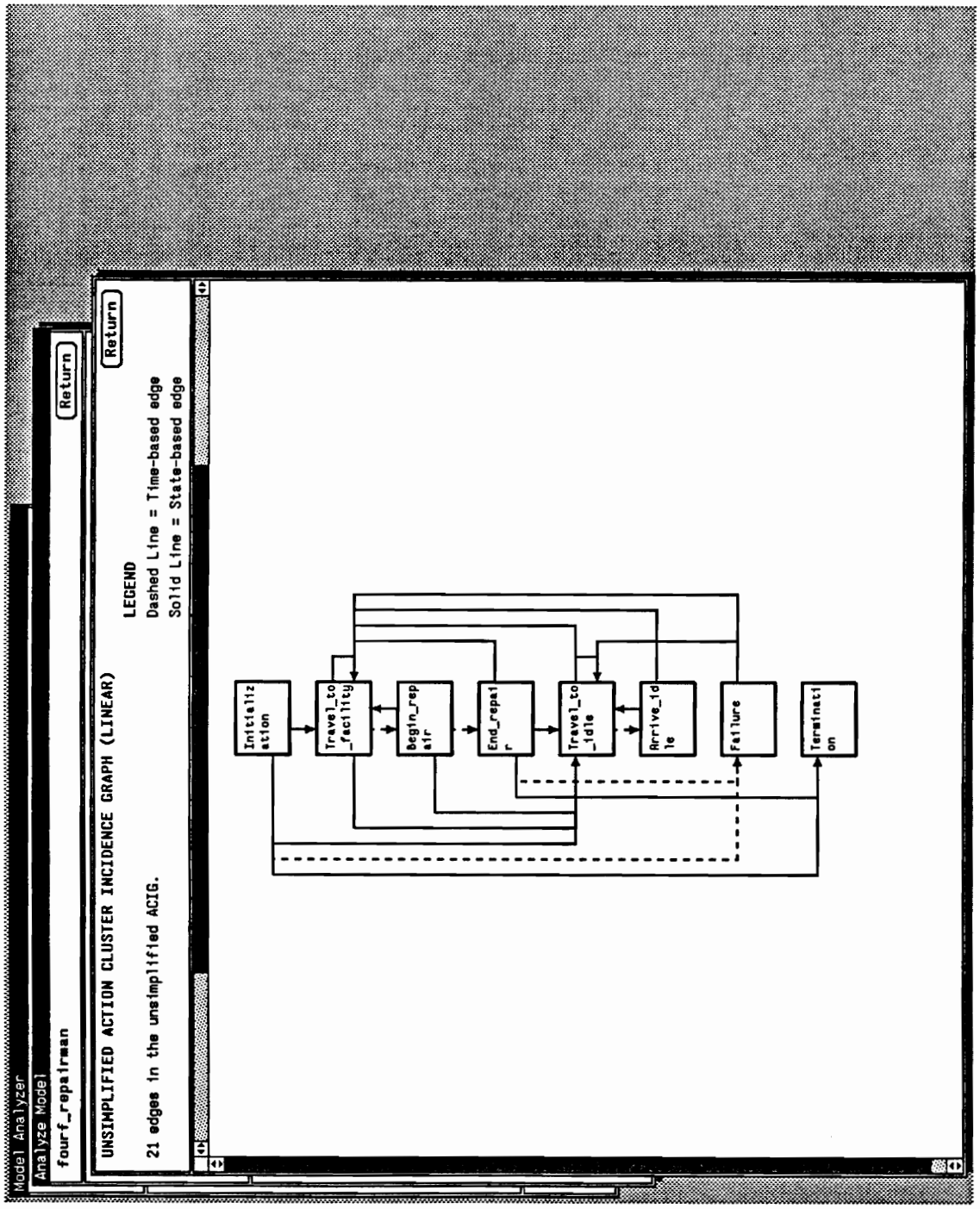


Figure 127. Unsimplified First Failed, First Fixed Machine Repairman Action Cluster Incidence Graph

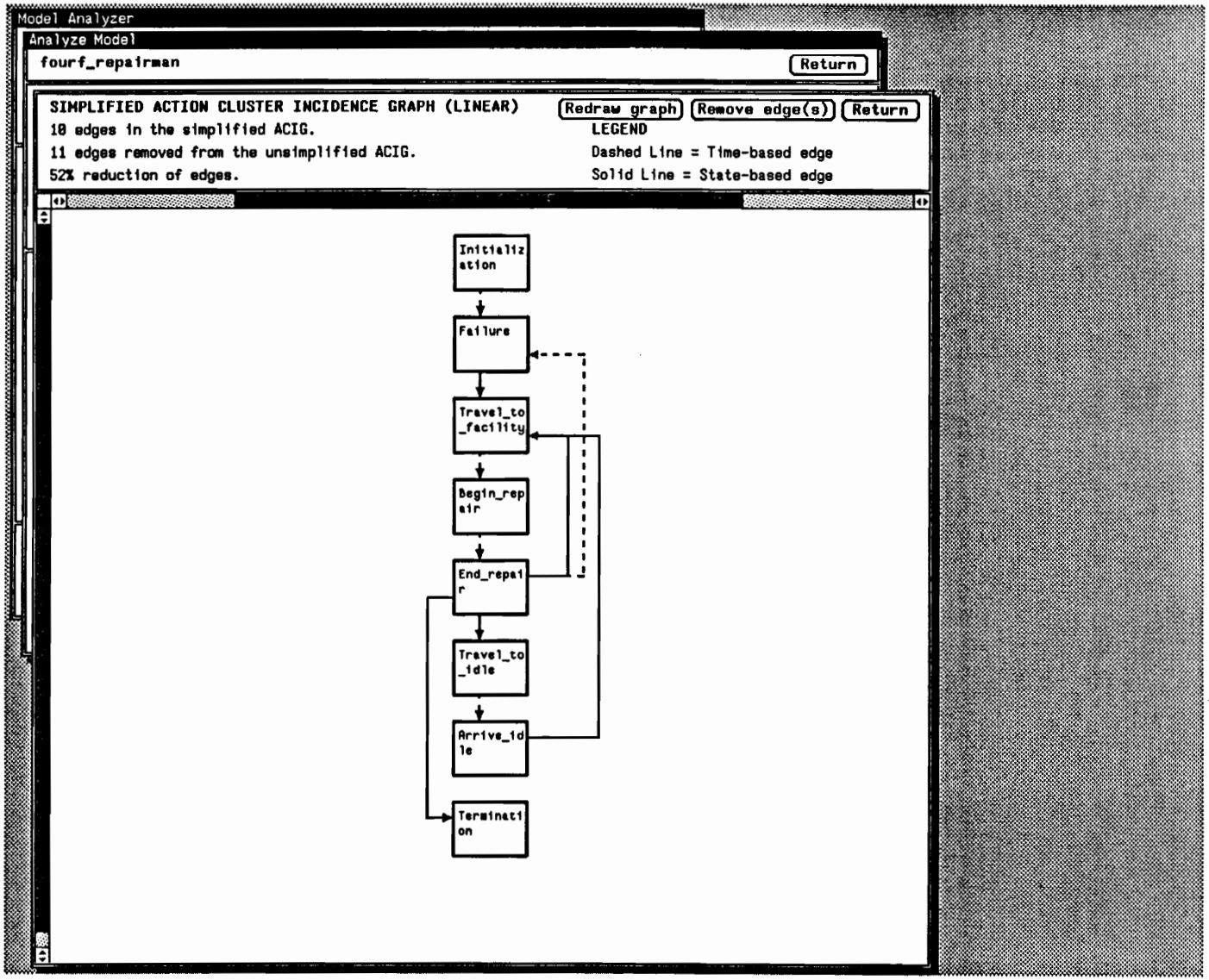


Figure 128. Simplified First Failed, First Fixed Machine Repairman Action Cluster Incidence Graph

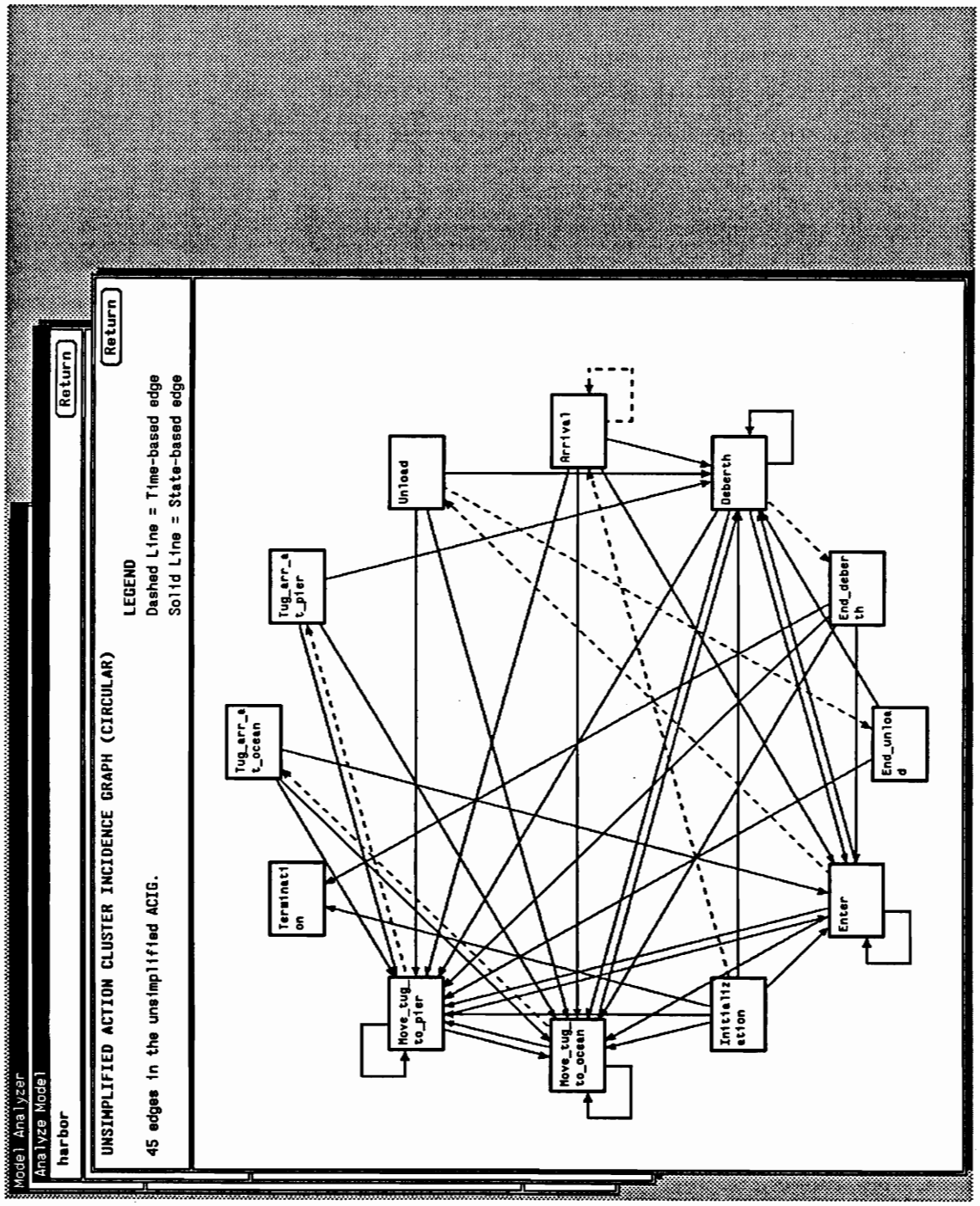


Figure 129. Unsimplified Harbor Model Action Cluster Incidence Graph

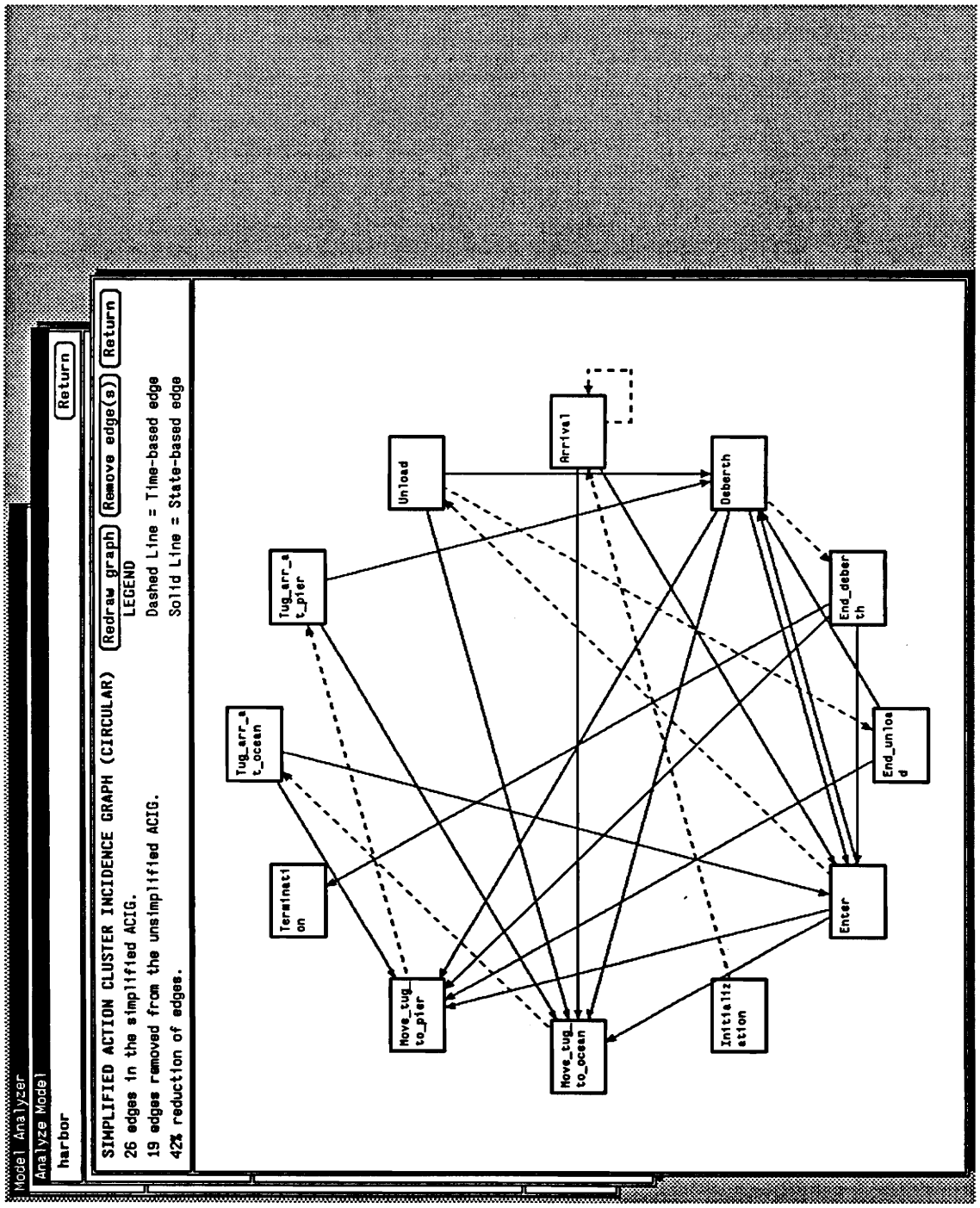


Figure 130. Simplified Harbor Model Action Cluster Incidence Graph

Appendix H. Model Analyzer User's Manual

The Model Analyzer, a Simulation Model Development Environment (SMDE) tool that analyzes the specifications of a model, consists of more than 16,500 lines of C and EQUEL/C code under the SunView programming environment. This manual describes the Model Analyzer in a task-oriented fashion. For assistance using the SunView interface, such as moving and resizing windows or using scrollbars, refer to [Sun Microsystems 1988b, c].

The Model Analyzer is composed of two executable processes: (1) *analyzer* and (2) *analysis*. *Analyzer* is the main process that creates the interface, allows the modeler to select a model, calls the *analysis* process (which parses the Condition Specification (CS) of a model), and performs the analysis of the model. Two subdirectories are also needed to execute the Model Analyzer: (1) *csfiles* and (2) *outfiles*. Each file in the subdirectory *csfiles* contains the CS for a model, and each file in the subdirectory *outfiles* is produced by the analysis of the Model Analyzer and viewed in text subwindows.

H.1 Start the Model Analyzer

The Model Analyzer can be started two different ways: (1) form the SMDE or (2) from the SunView command line. First, the Model Analyzer is activated by selecting the "Model Analyzer" button, shown in Figure 131. Unless the user has proper priorities and access privileges, the Model Analyzer cannot be executed in this manner. Secondly, from within SunView, the Model Analyzer is activated by typing the word "analyzer" on

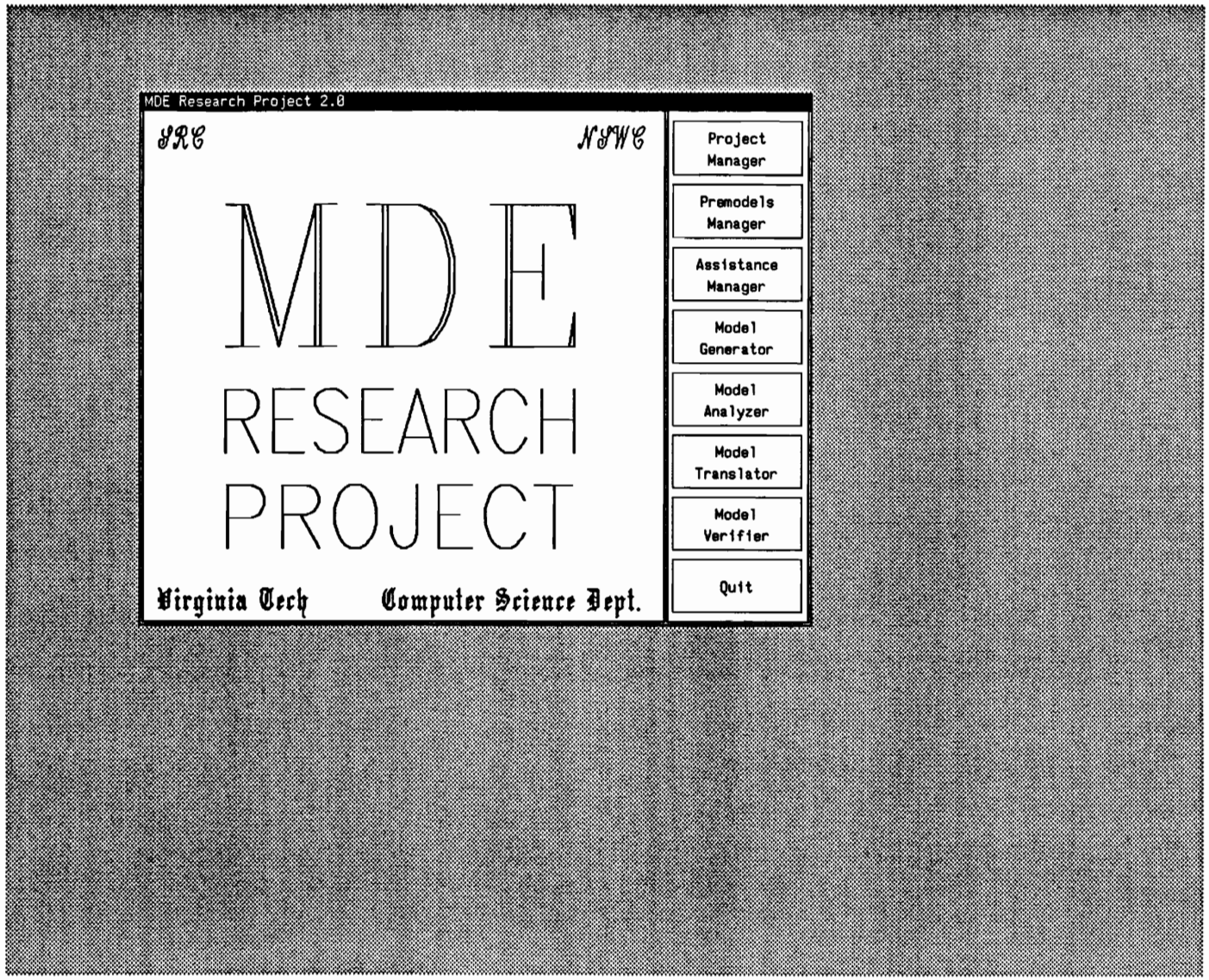


Figure 131. MA: The Simulation Model Development Environment

the command line while in the directory "usr/puthoff/analyzer". Figure 132 shows the top level of the Model Analyzer after activation.

H.2 Exit the Model Analyzer

To terminate the execution of the Model Analyzer, select the icon that looks like a stop sign. After selecting the icon, an alert message appears requesting the modeler to quit, shown in Figure 133. To quit, select the "Confirm" button.

H.3 Select a Model

To select a model to be analyzed, use the mouse to select a model name from the list in the right hand side of the Model Analyzer frame. When a model name is selected, that name appears to the right of the words "Model name:".

H.4 View and Edit the Condition Specification of a Model

To view the CS of the selected model, select the icon that resembles a huge eyeball. When the icon is selected, the Model Viewer frame, shown in Figure 134, appears containing the CS for the selected model.

To edit the CS in the Model Viewer frame, select the "Edit" button. After selecting the "Edit" button, an editing cursor the shape of a small triangle appears in the CS. Use the editing cursor to make the necessary changes in the CS.

After editing the CS, the user can either (1) save the editing changes or (2) disregard the editing changes. To save the editing changes to the CS, select the "Return/Save" button, and to ignore the editing changes, select the "Quit/No Save" button. After selecting one of the two buttons, the user is returned to the Model Analyzer frame.

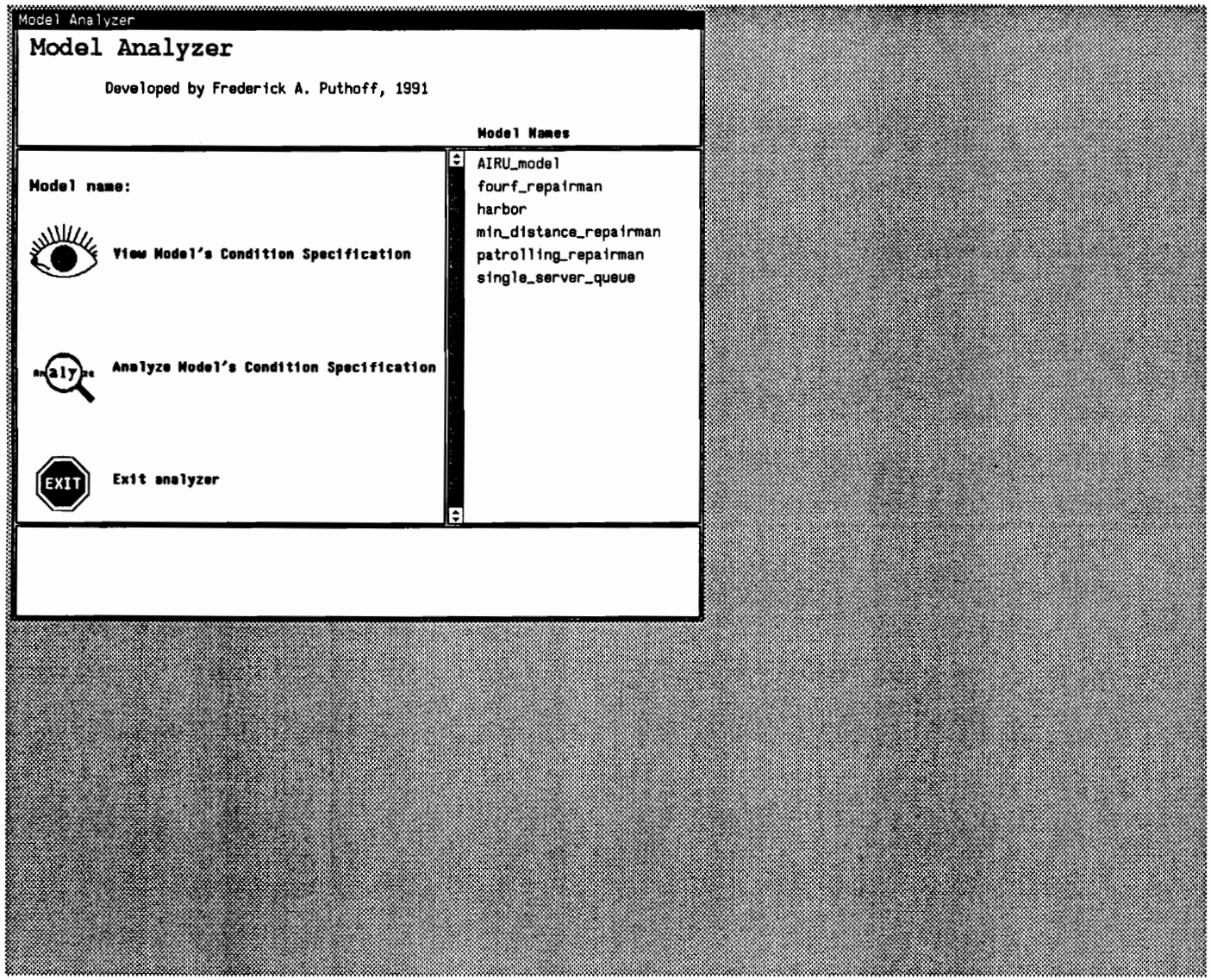


Figure 132. MA: The Top Level of the Model Analyzer

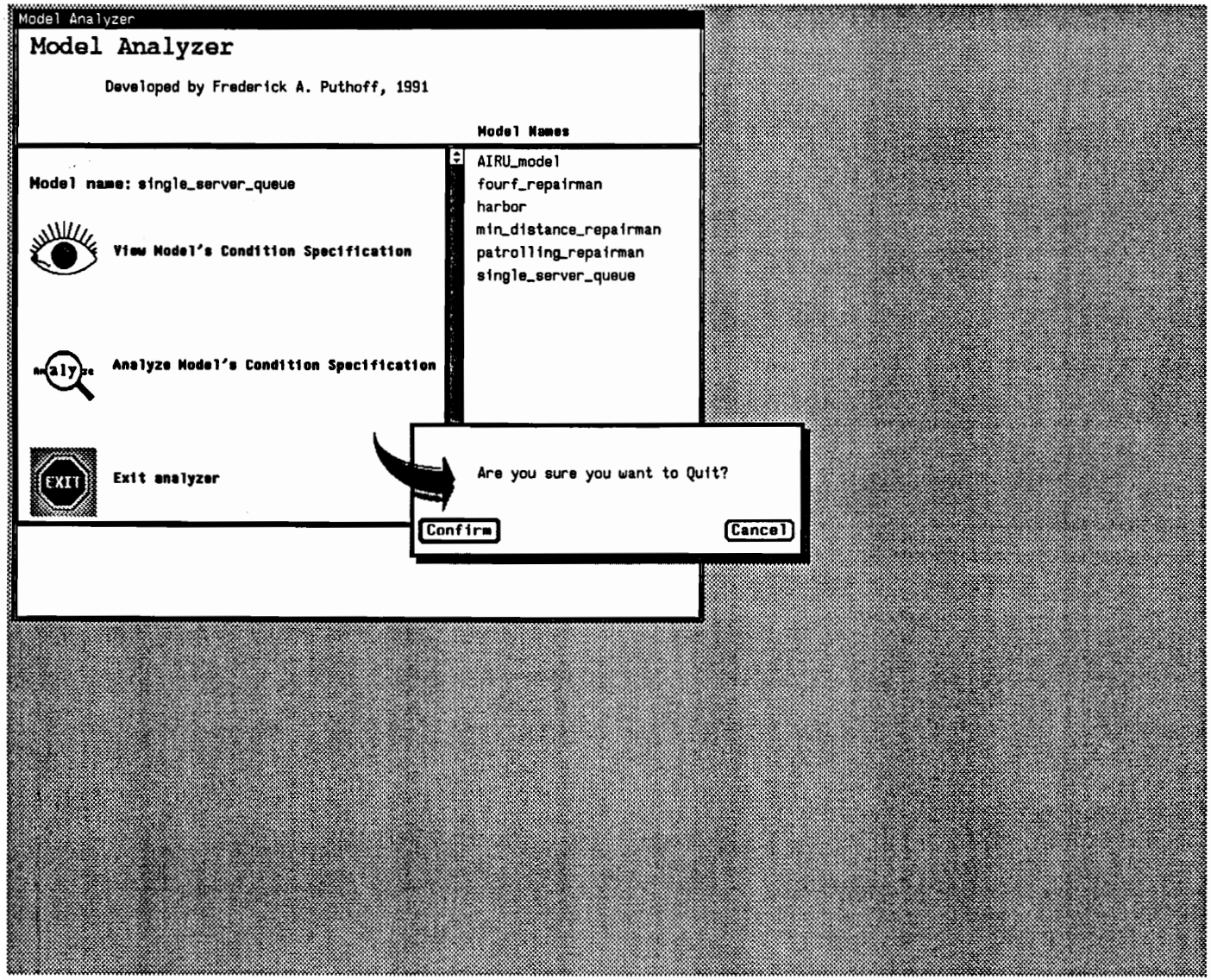


Figure 133. MA: Exit the Model Analyzer

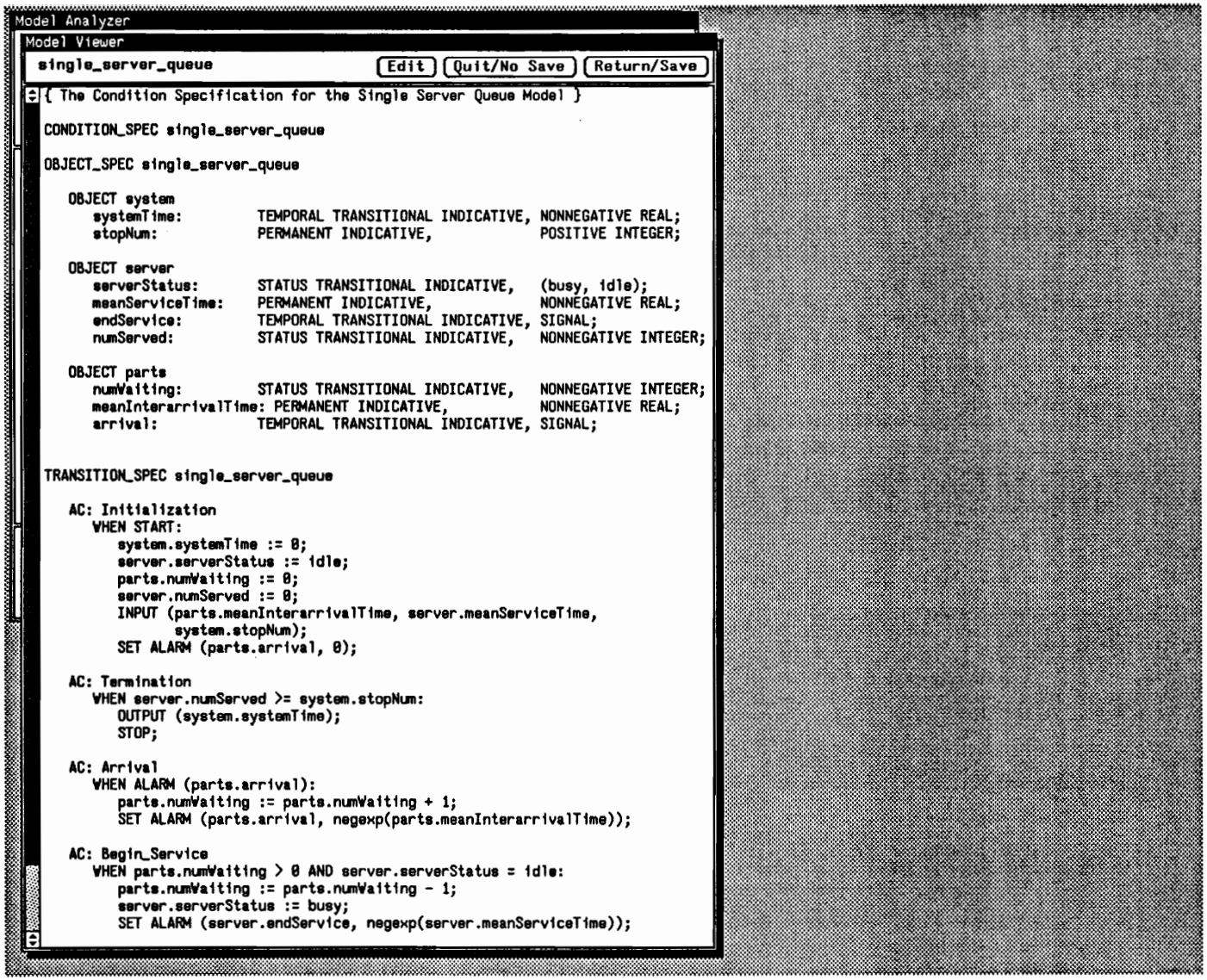


Figure 134. MA: View the Condition Specification of a Model

H.5 Analyze the Condition Specification of a Model

To analyze the specifications of the selected model, depress the icon that resembles a magnifying glass in the Model Analyzer panel. After selecting the icon, the cursor changes to an hourglass and a message appears asking the modeler to wait as the Model Analyzer prepares the model to be analyzed (Figure 135). If an error is found while parsing the CS of the model, an error message is displayed, shown in Figure 136, and the analysis preparation stops. If no errors are found, the Analyze Model frame, shown in Figure 137, is displayed after a wait of approximately two to five minutes.

The model is ready to be analyzed when the Analyze Model frame appears. The analysis session is broken into three major functions: (1) listing various aspects of the CS of the model, (2) viewing the graphical representations of the model, (3) and performing diagnostic assistance upon the model specifications. The following sections describe each of these three functions and how to terminate the analysis session.

H.5.1 Quit Analysis Session

To quit the analysis session from the Analyze Model frame, select the "Return" button. After selecting this button, an alert message appears, shown in Figure 138, that asks if the modeler wishes to end the analysis session. Selecting the "Confirm, return" button ends the analysis session and brings the modeler back to the Model Analyzer frame.

H.5.2 List Aspects of the Condition Specification

The top panel of the Analyze Model frame contains six buttons that list various aspects of the CS of a model. The aspects are described in the following seven subsections.

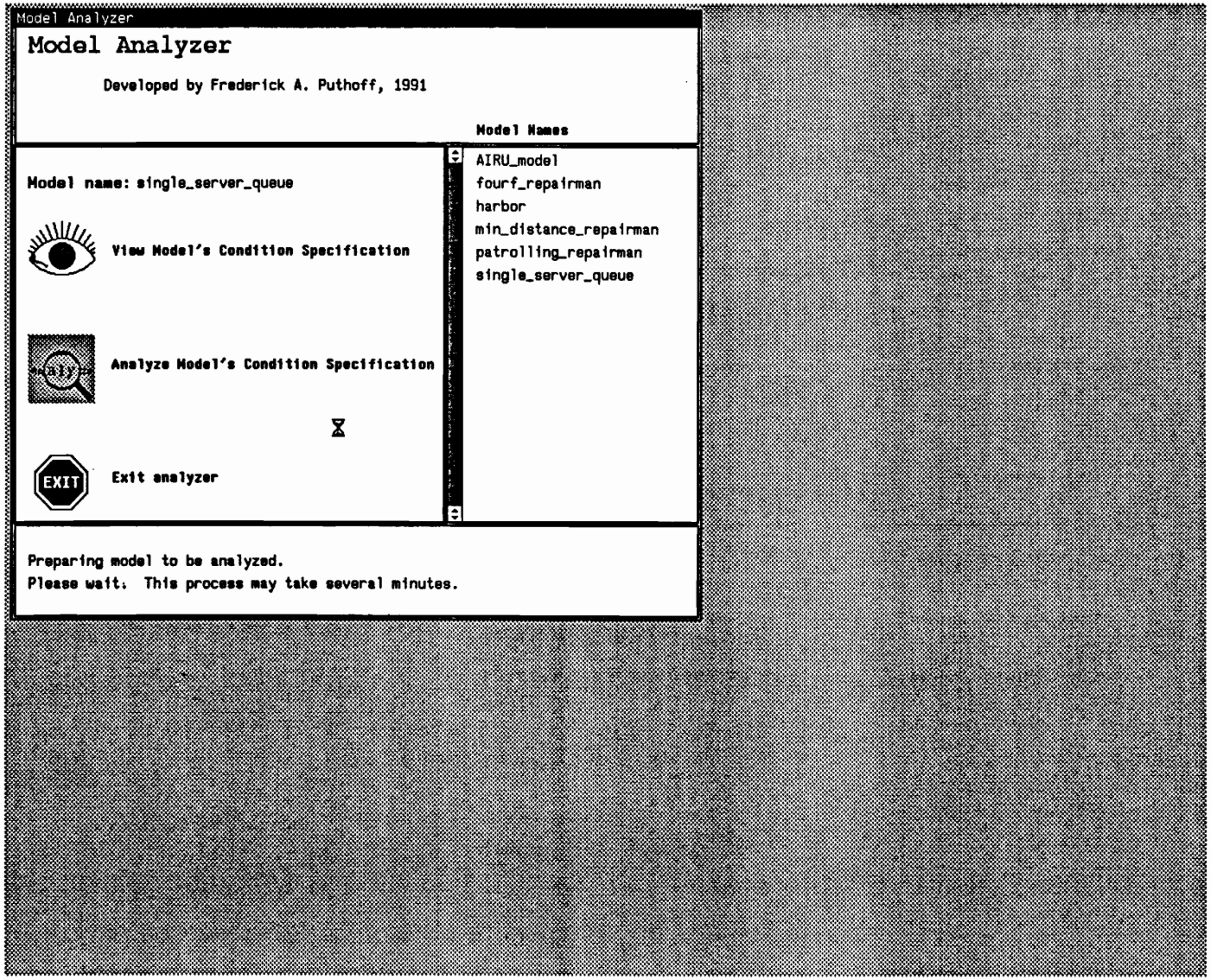


Figure 135. MA: Preparing to Analyze the Model

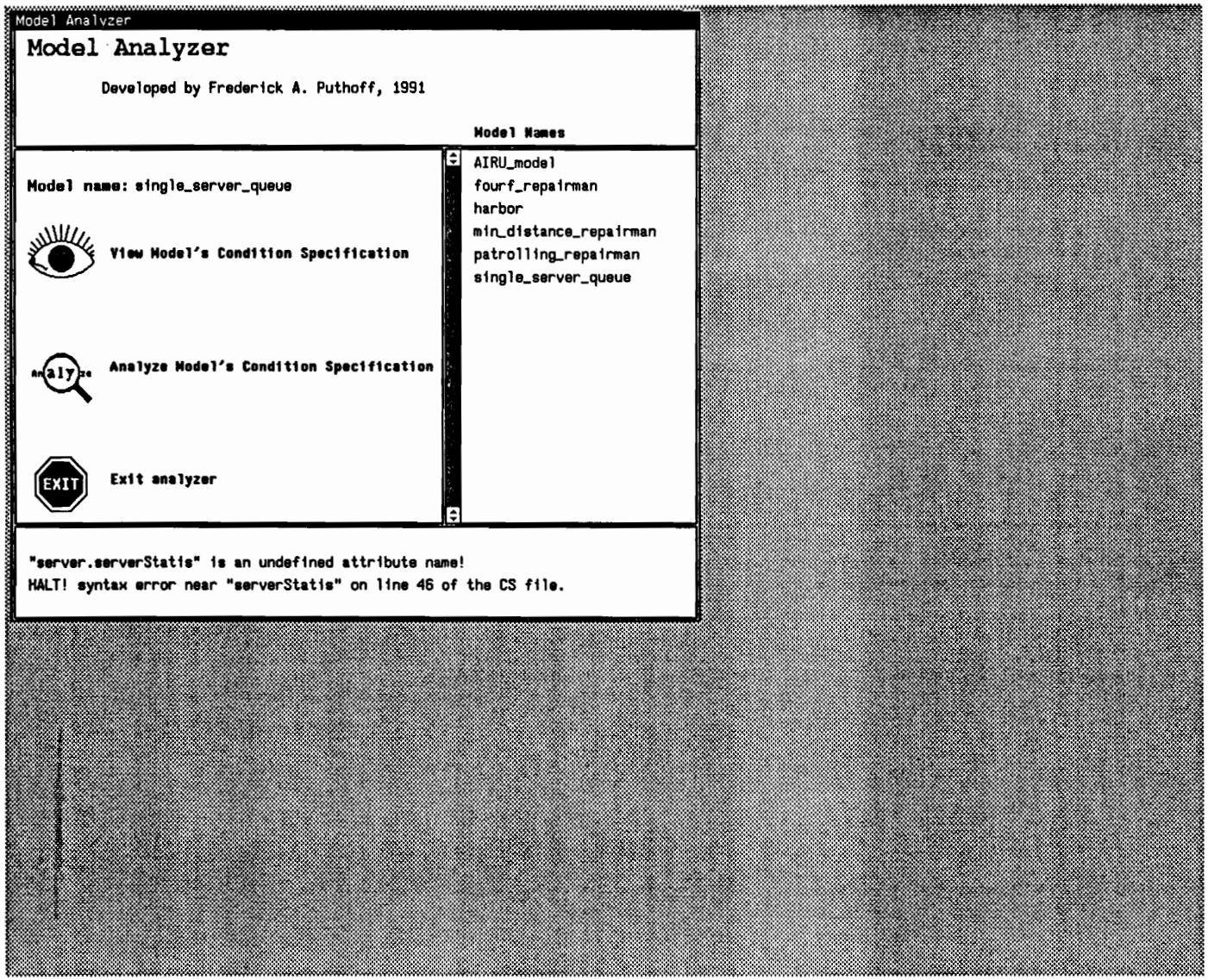


Figure 136. MA: An Error Found While Parsing the Condition Specification

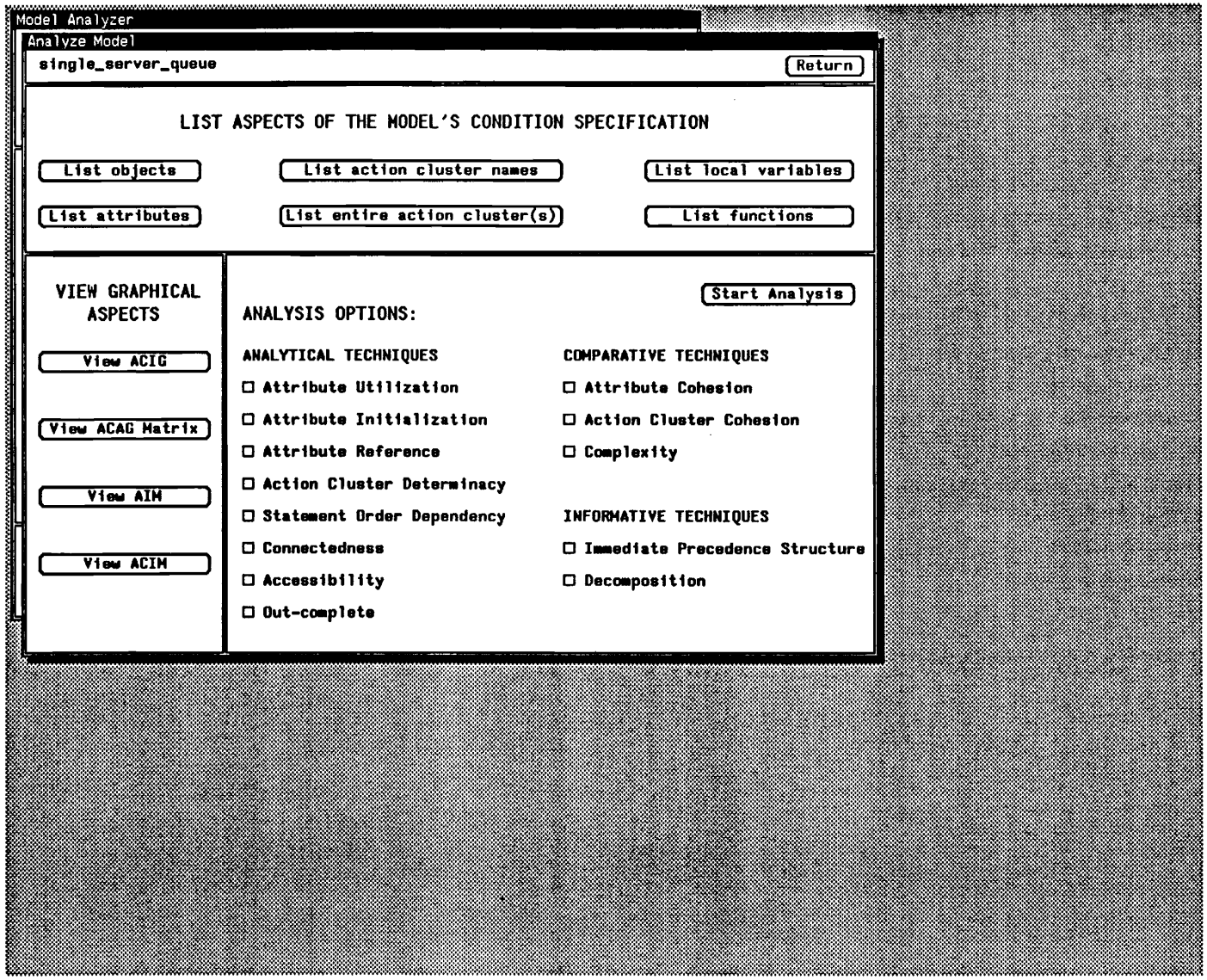


Figure 137. MA: The Analyze Model Frame in the Model Analyzer

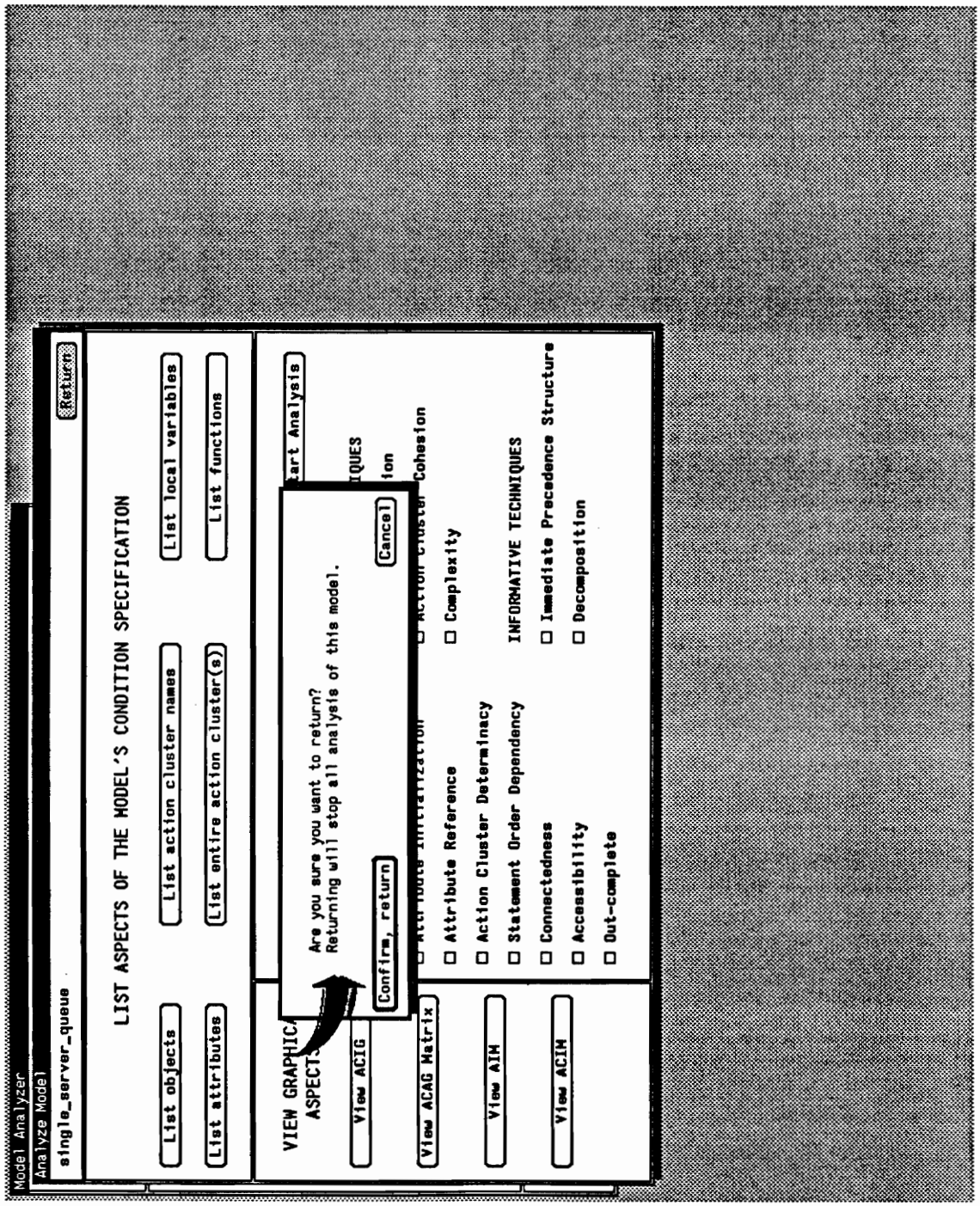


Figure 138. MA: Quit the Analysis Session

H.5.2.1 List Model Objects

To list the objects of the model, select the "List objects" button. Selecting the button displays a panel that lists the model objects, shown in Figure 139.

H.5.2.2 List Action Cluster Names

To list the names of the action clusters in the CS, select the "List action cluster names" button. When the button is selected, a panel appears, shown in Figure 140, listing the action clusters of the CS.

H.5.2.3 List Attributes of Objects

The modeler can list the Conical Methodology (CM) indicative, CM relational, or all attributes of one or all model objects by pulling right on the "Of Object(s)" menu item after selecting the "List attributes" button. Figure 141 shows the series of pull right menus associated with the "Of Object(s)" menu item.

To view the indicative attributes of one or all model objects, select the "List attributes" button, pull right on the "Of Object(s)" item in the "ATTRIBUTES:" menu, pull right on the "Indicative" item in the "ATTRIBUTE TYPE:" menu, and select an item in the "FOR OBJECT:" menu. Selecting the "All objects" item displays a panel that lists the indicative attributes of all model objects, and selecting any other item in the menu displays a panel that lists the indicative attributes of the selected object.

To view the relational attributes of one or all model objects, follow the same procedures as viewing the indicative attributes except pull right on the "Relational" item instead of the "Indicative" item in the "ATTRIBUTE TYPE:" menu.

To list all attributes of one or all model objects, select the "List attributes" button, pull right on the "Of Object(s)" item in the "ATTRIBUTES:" menu, pull right on the

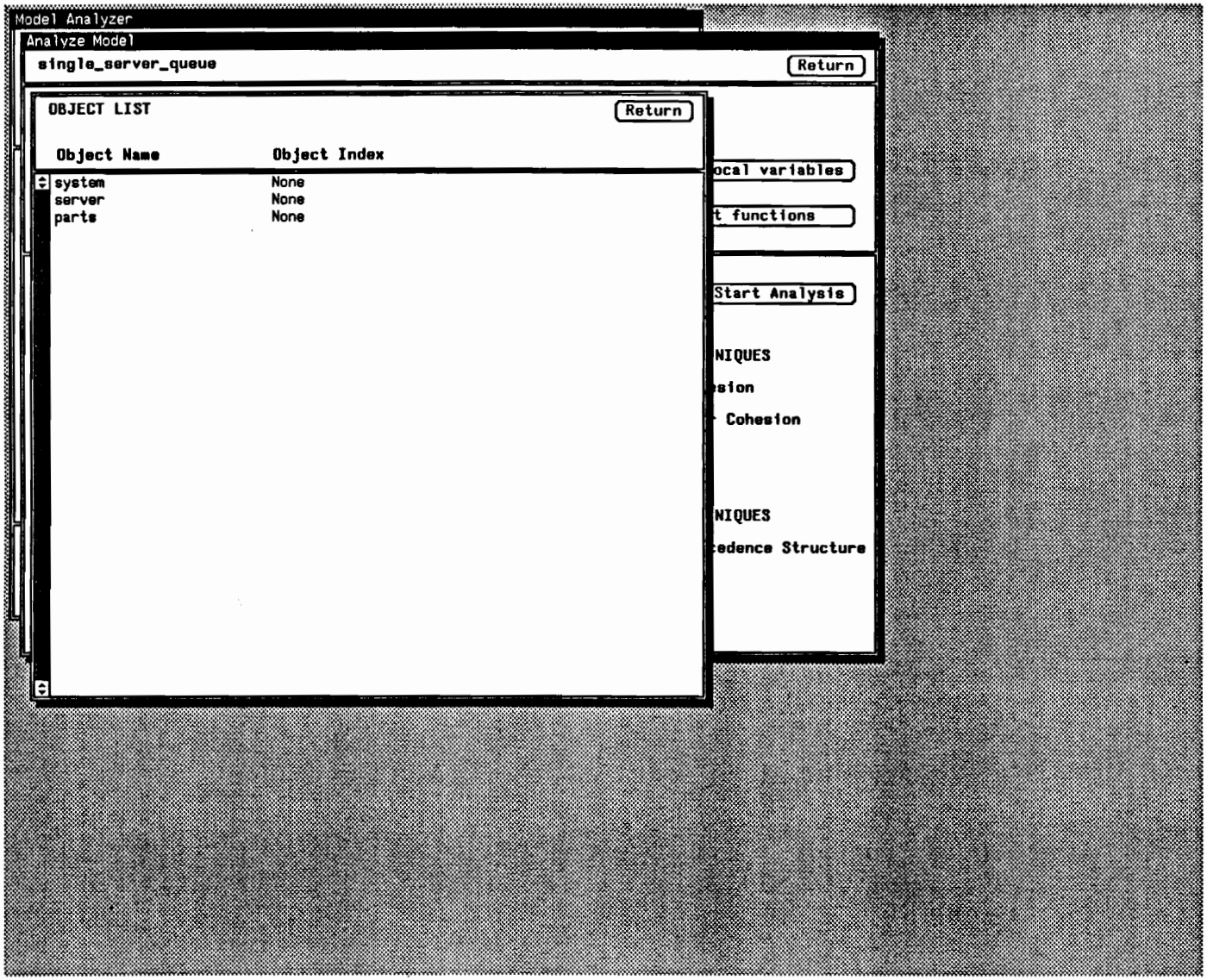


Figure 139. MA: List the Model Objects

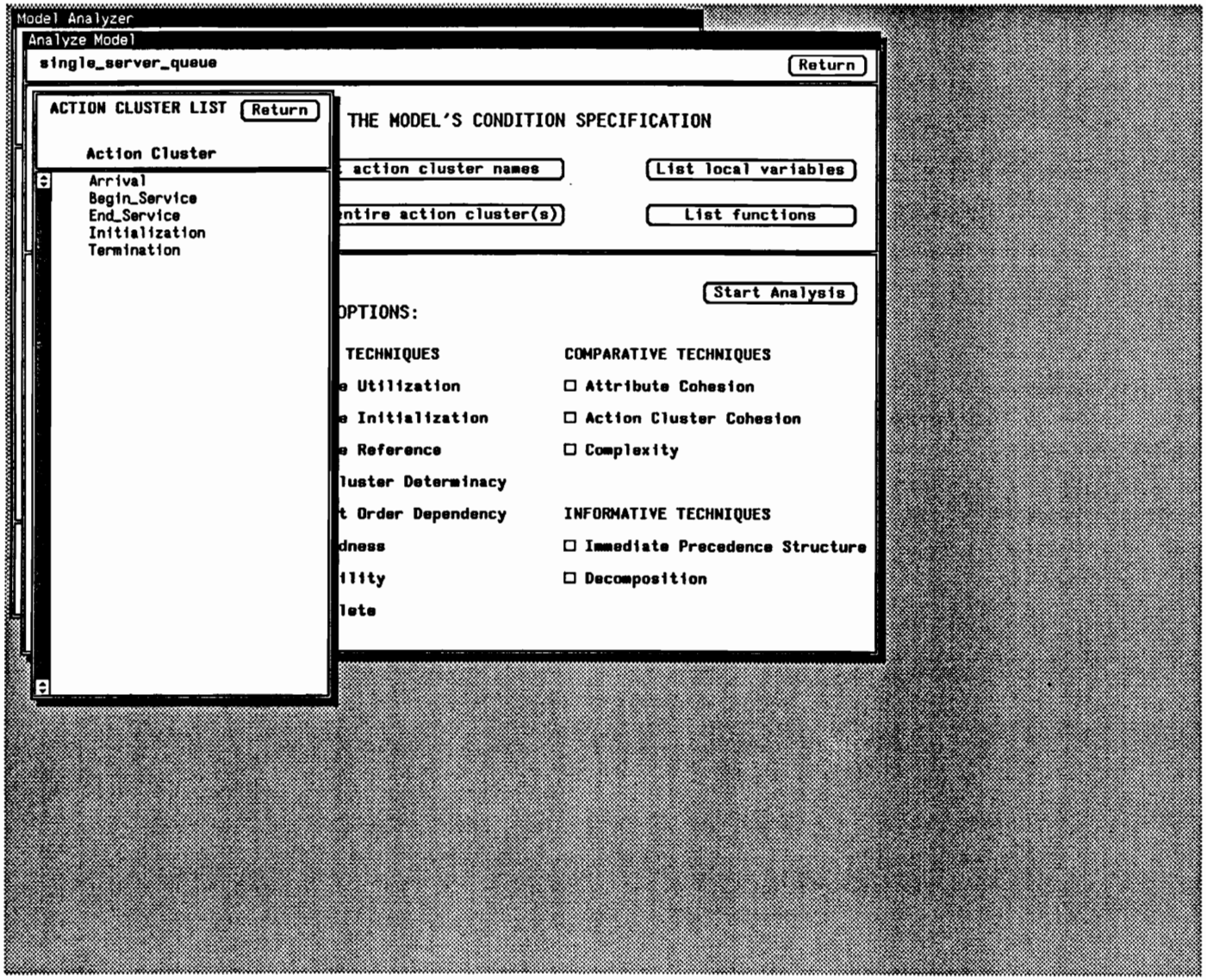


Figure 140. MA: List the Action Cluster Names

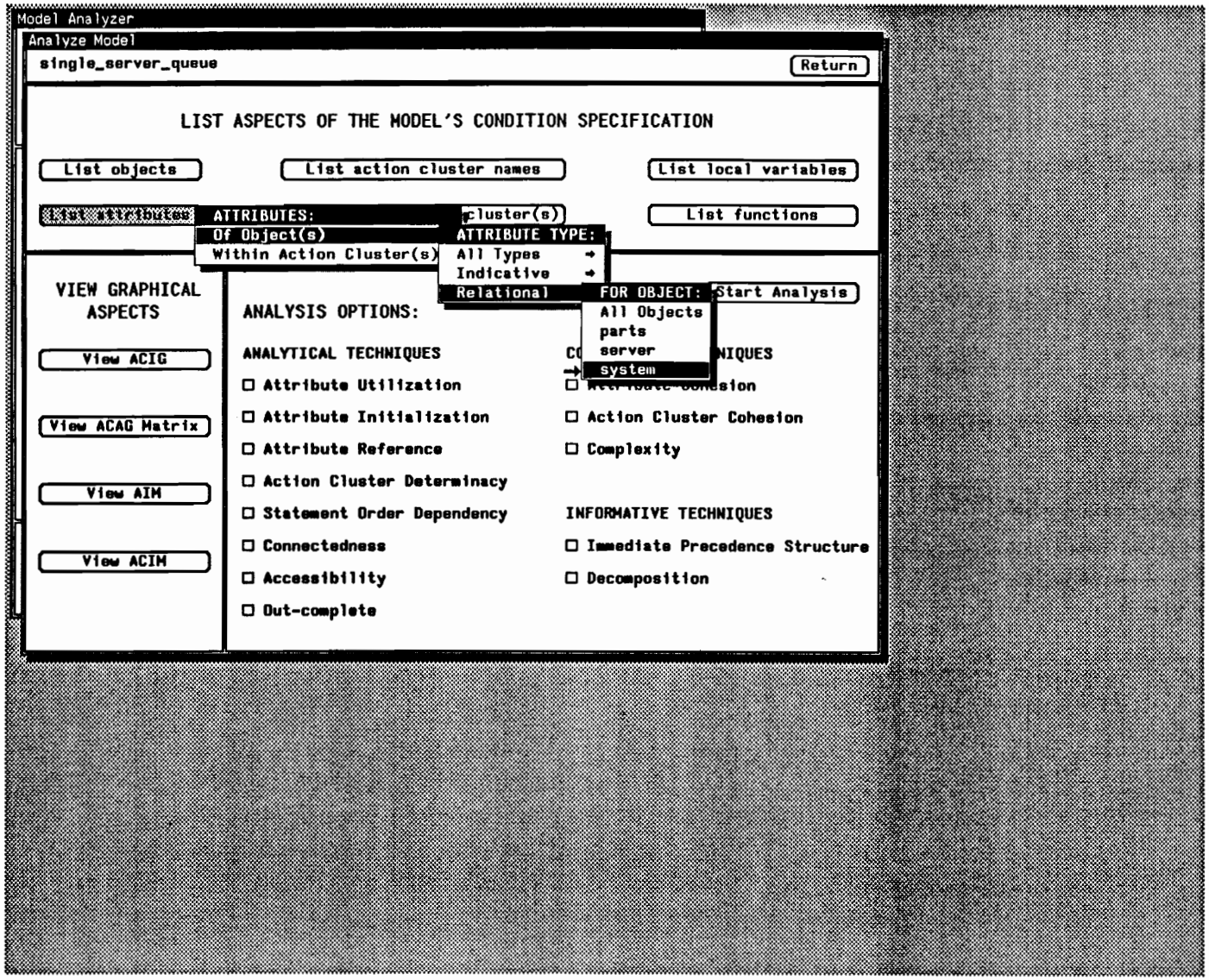


Figure 141. MA: The "Attributes: Of Object(s)" Series of Menus

"All Types" item in the "ATTRIBUTE TYPE:" menu, and select an item in the "FOR OBJECT:" menu. Selecting the "All objects" item lists and defines by CM and CS type all attributes of all model objects (Figure 142), and selecting any other item in the menu lists and defines all attributes of the selected object.

H.5.2.4 List Attribute Classification in Action Clusters

By pulling right on the "Within Action Cluster(s)" menu item after selecting the "List attributes" button, the modeler can list the classification (input, output, or control) of each attribute in one or all action clusters of the CS. Figure 143 shows the series of pull right menus associated with the "Within Action Cluster(s)" menu item.

To view the input, output, or control attributes of one or all action clusters, select the "List attributes" button, pull right on the "Within Action Cluster(s)" item in the "ATTRIBUTES:" menu, pull right on the "Input", "Output", or "Control" item in the "ATTRIBUTE TYPE:" menu, and select an item in the "FOR ACTION CLUSTER:" menu. Selecting the "All action clusters" item displays a panel that lists the input, output, or control attributes of all action clusters, and selecting any other item in the menu displays a panel that lists the input, output, or control attributes of the selected action cluster.

To view all (input, output, and control) attributes of one or all action clusters, follow the same procedure as for listing the input, output, or control attributes but pull right on the "All Types" item instead of any of the other three items in the ATTRIBUTE TYPE:" menu. Figure 144 shows a panel, which lists all attributes of all action clusters in the CS.

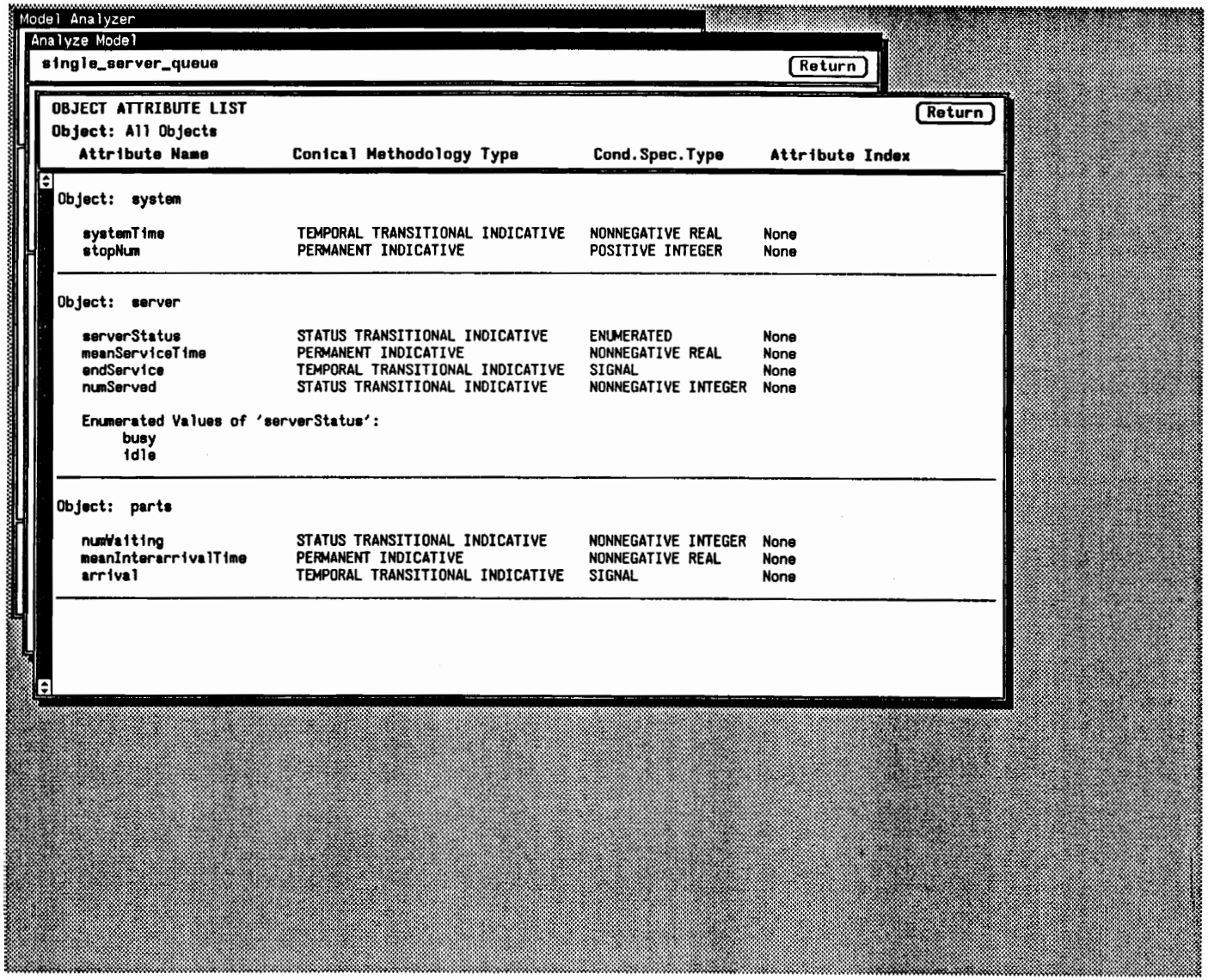


Figure 142. MA: List All Attributes of All Objects

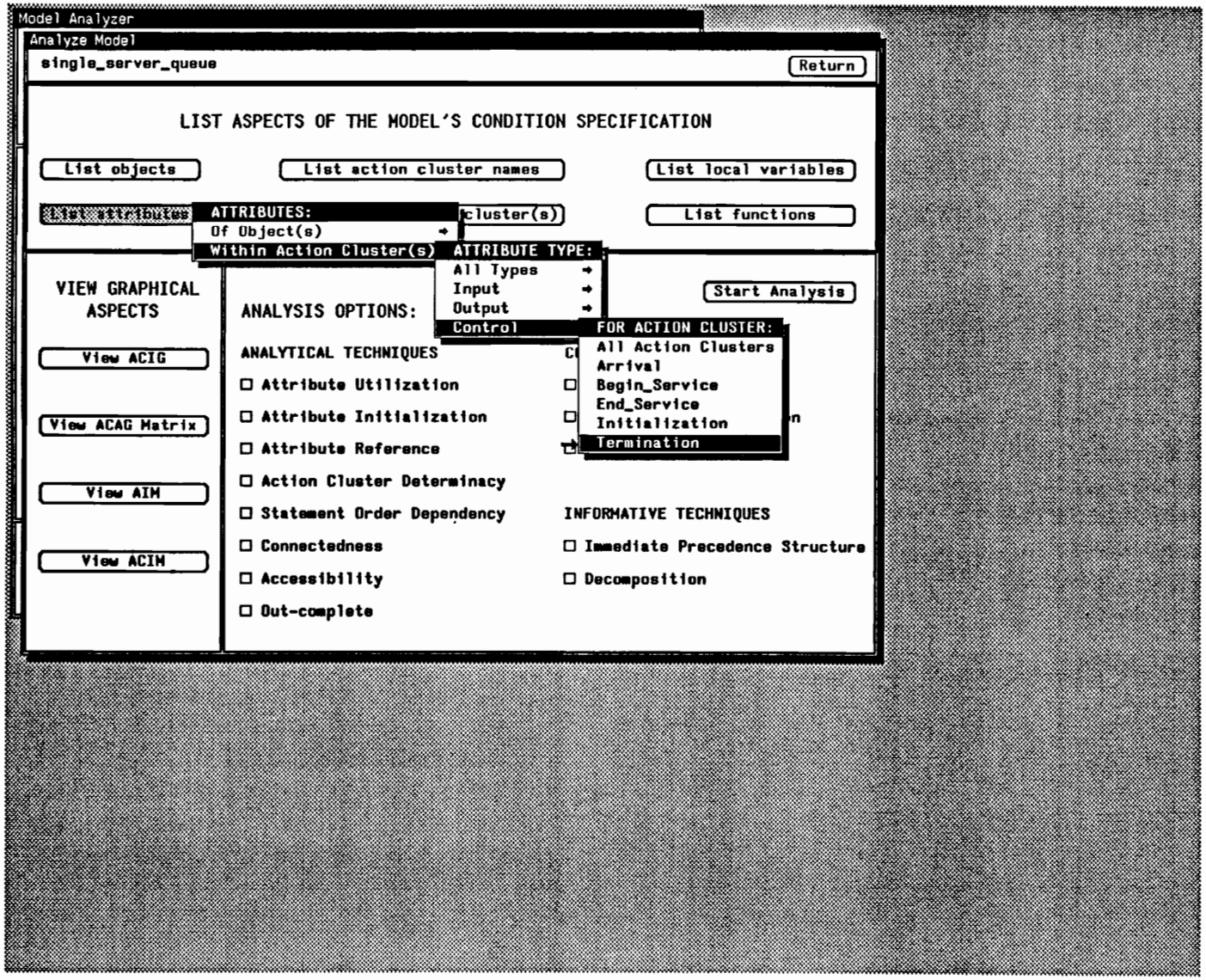


Figure 143. MA: The "Attributes: Within Action Cluster(s)" Series of Menus

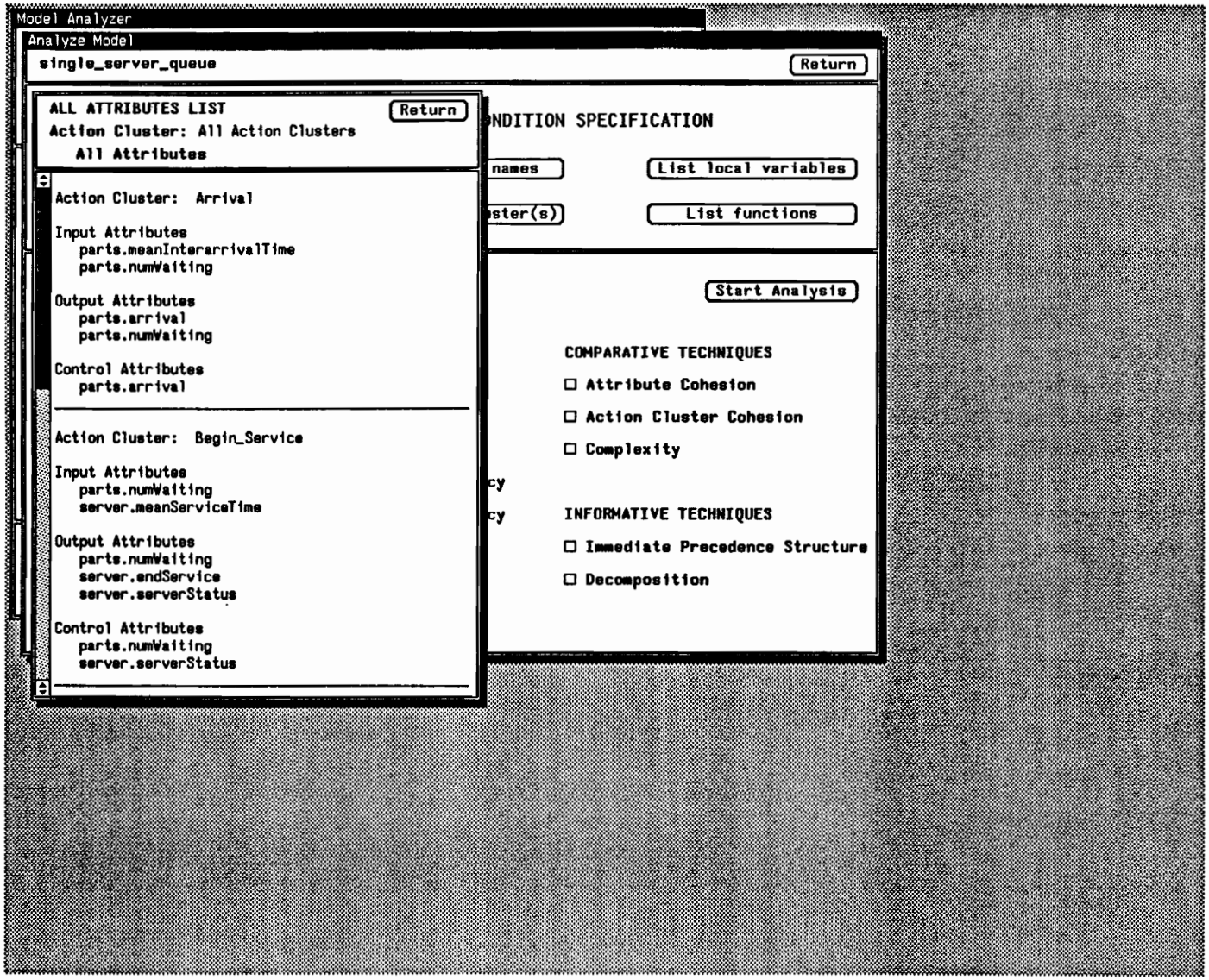


Figure 144. MA: List the Classification of All Attributes in All Action Clusters

H.5.2.5 List Contents of Action Clusters

To list the contents (condition and actions) of one or all action clusters in the CS, select the "List entire action cluster(s)" button. When the button is selected, a menu, shown in Figure 145, appears that lists all the action clusters in the CS and the item "All Action Clusters". Selecting the "All Action Clusters" item displays the conditions and actions of all action clusters, and selecting any other item in the menu displays the condition and actions of the selected action cluster, shown in Figure 146.

H.5.2.6 List Local Variables in Action Clusters

Select the "List local variables" button to list the local variables of one or all action clusters in the CS. When the button is selected, a menu, identical with the menu of the "List entire action cluster(s)" button shown in Figure 145, displays all the action clusters in the CS and the item "All Action Clusters". Selecting the "All Action Clusters" item lists the local variables used in each action cluster, and selecting any other item in the menu displays the local variables used in the selected action cluster, shown in Figure 147.

H.5.2.7 List Functions in Action Clusters

To list the functions of one or all action clusters in the CS, select the "List functions" button. When the button is selected, a menu, identical with the menu of the "List entire action cluster(s)" button shown in Figure 145, displays all the action clusters in the CS and the item "All Action Clusters". Selecting the "All Action Clusters" item lists the functions used by each action cluster, shown in Figure 148, and selecting any other item in the menu displays the functions used by the selected action cluster.

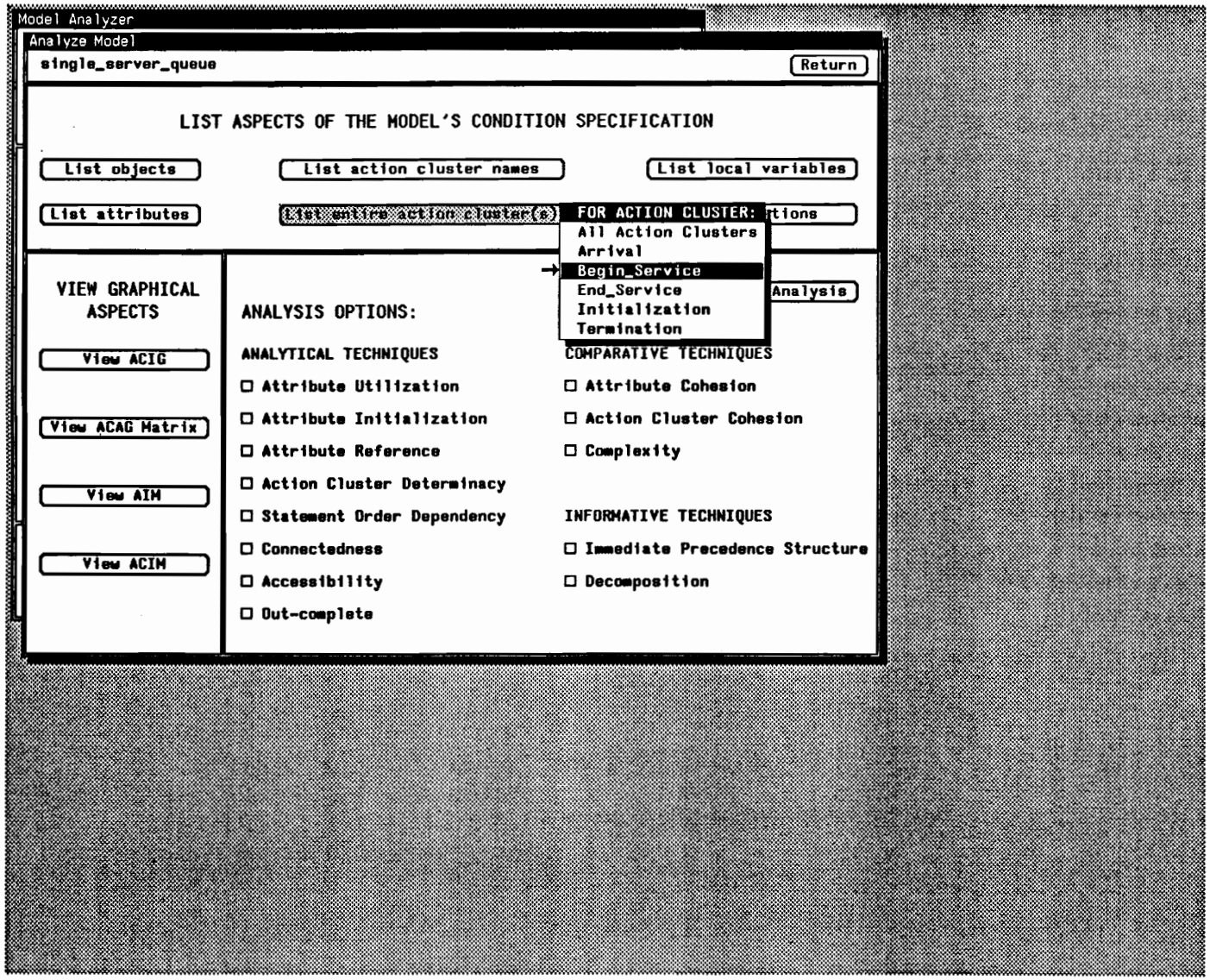


Figure 145. MA: The Menu of the "List entire action cluster(s)" Button

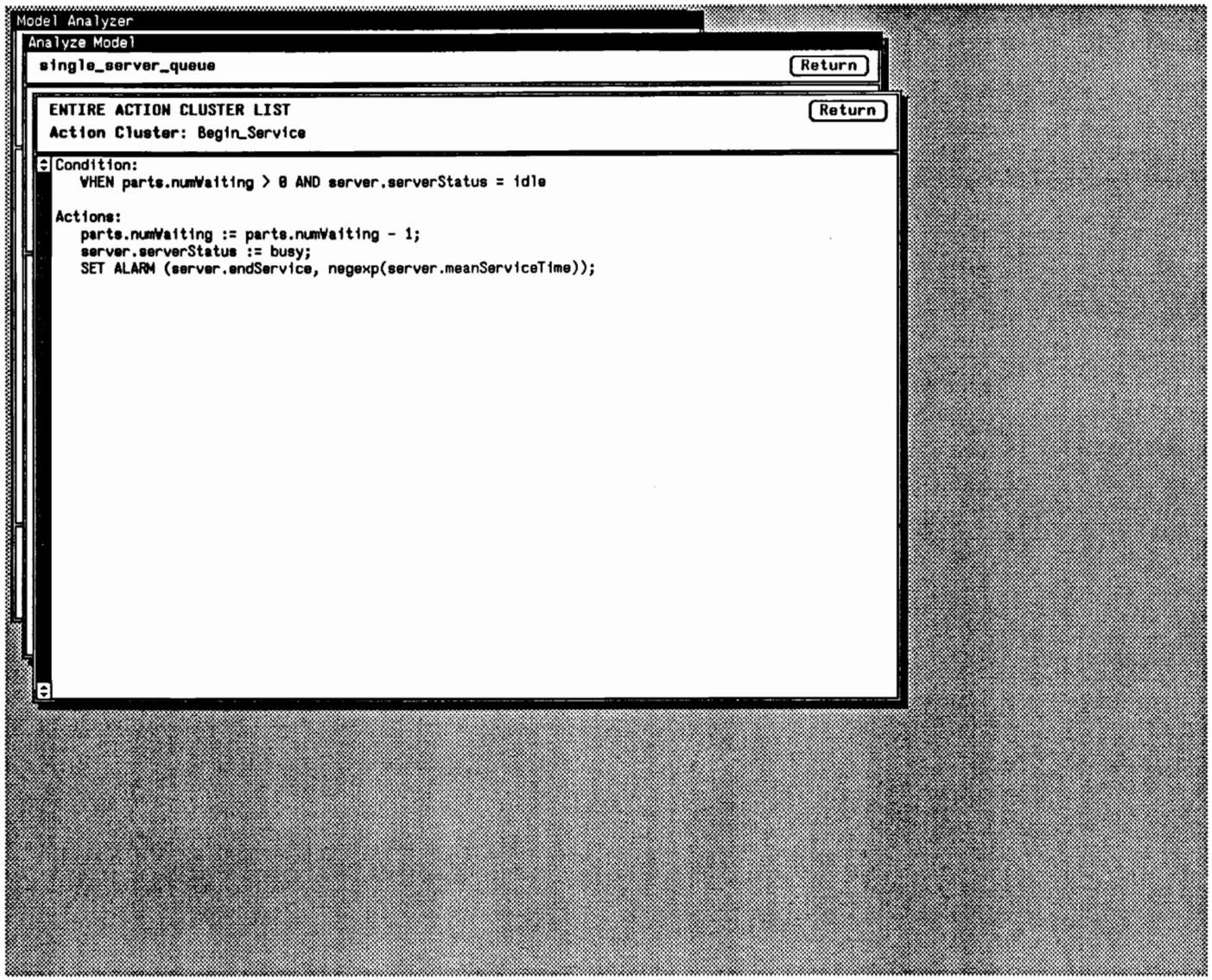


Figure 146. MA: List the Contents of An Action Cluster

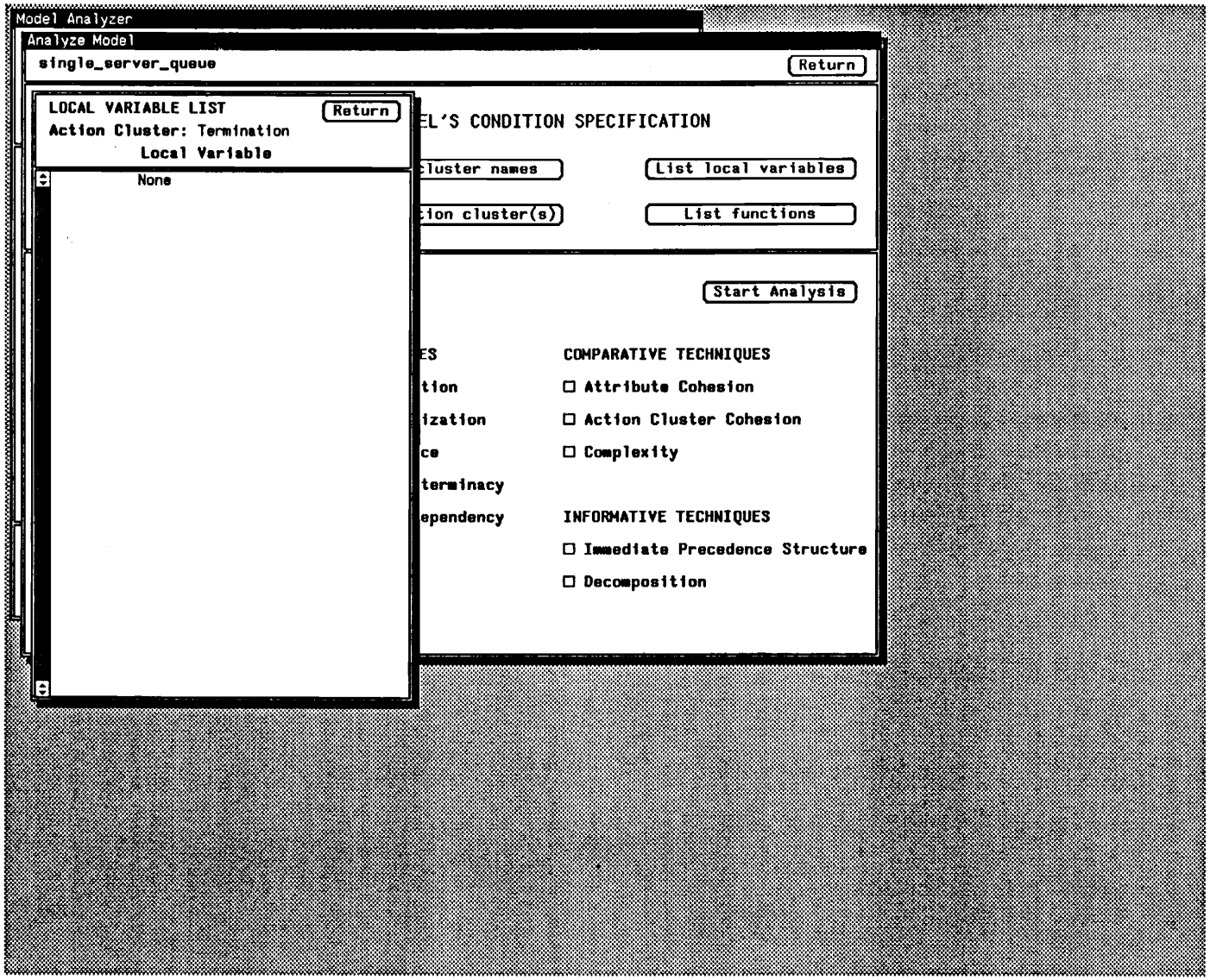


Figure 147. MA: List the Local Variables Used in an Action Cluster

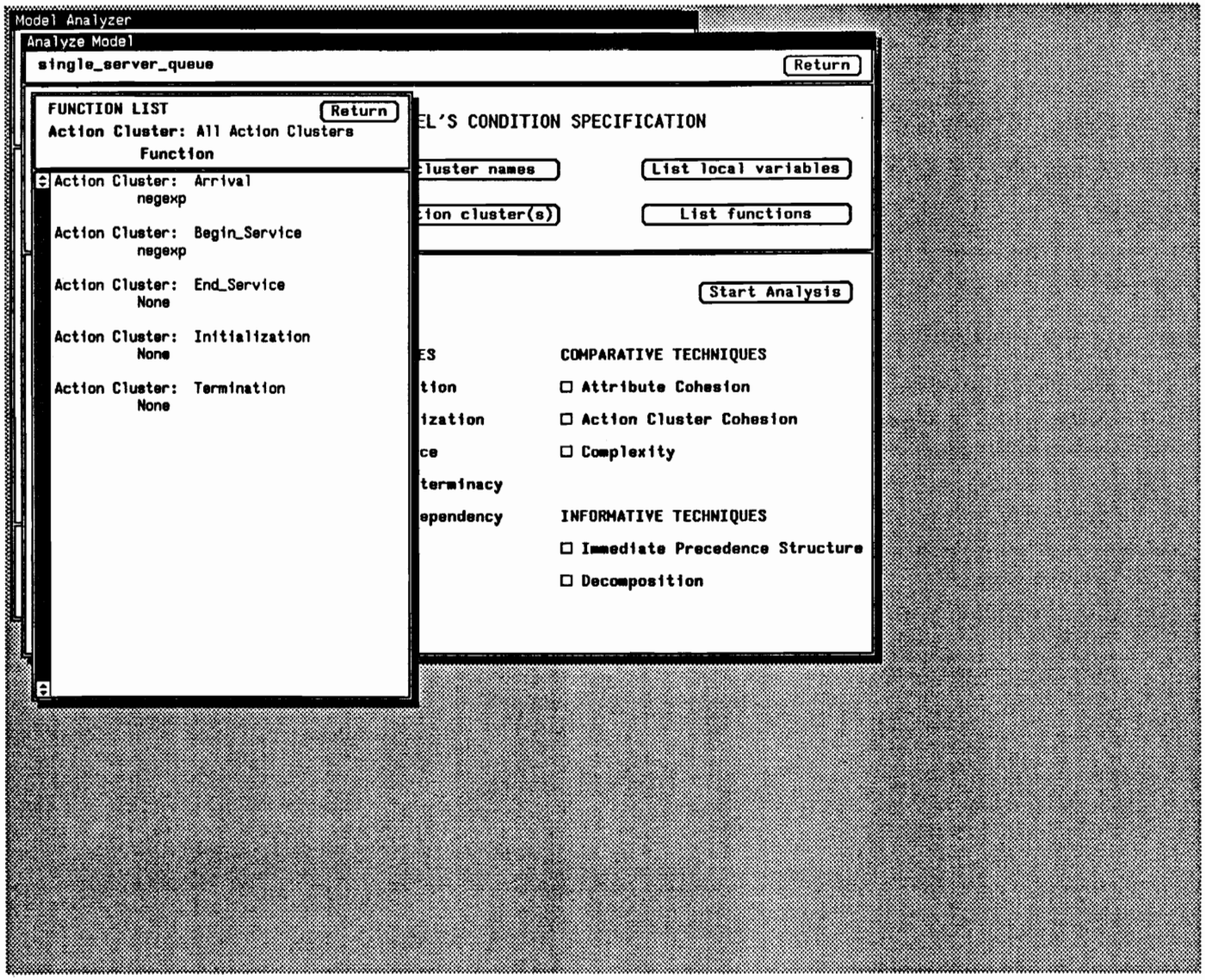


Figure 148. MA: List the Functions Used in Each Action Cluster

H.5.3 View Graphical Features of the Condition Specification

The four buttons located in the bottom left panel of the Analyze Model allow the modeler to view four graphical representations of a model. The following four subsections describe how to view each representation.

H.5.3.1 View Action Cluster Incidence Graph

To view the Action Cluster Incidence Graph (ACIG) of a model, select the "View ACIG" button. The ACIG is viewed as a matrix, circular graph, or linear graph in an unsimplified or simplified fashion, as shown by the menus associated with the "View ACIG" button in Figure 149.

H.5.3.1.1 View Unsimplified Action Cluster Incidence Graph

To view the unsimplified ACIG of a model, select the "View ACIG" button, pull right on the "Unsimplified" item in the "ACIG TYPE:" menu, and select one of the three items in the "ACIG FORM:" menu. Selecting the "Matrix" menu item displays the unsimplified ACIG as a matrix, shown in Figure 150. To view the unsimplified ACIG as a graph with the action cluster nodes arranged in a circle (Figure 151), select the "Circular Graph" button. Selecting the "Linear Graph" menu item displays the unsimplified ACIG as a graph with the action cluster nodes arranged in a vertical line.

H.5.3.1.2 View Simplified Action Cluster Incidence Graph

To view the simplified ACIG follow the same steps as viewing the unsimplified ACIG but pull right on the "Simplified" item instead of the "Unsimplified" item in the "ACIG TYPE:" menu. The simplified ACIG is also viewed as a matrix, circular graph, or linear graph. Figure 152 shows the simplified ACIG in the matrix form. Note that the

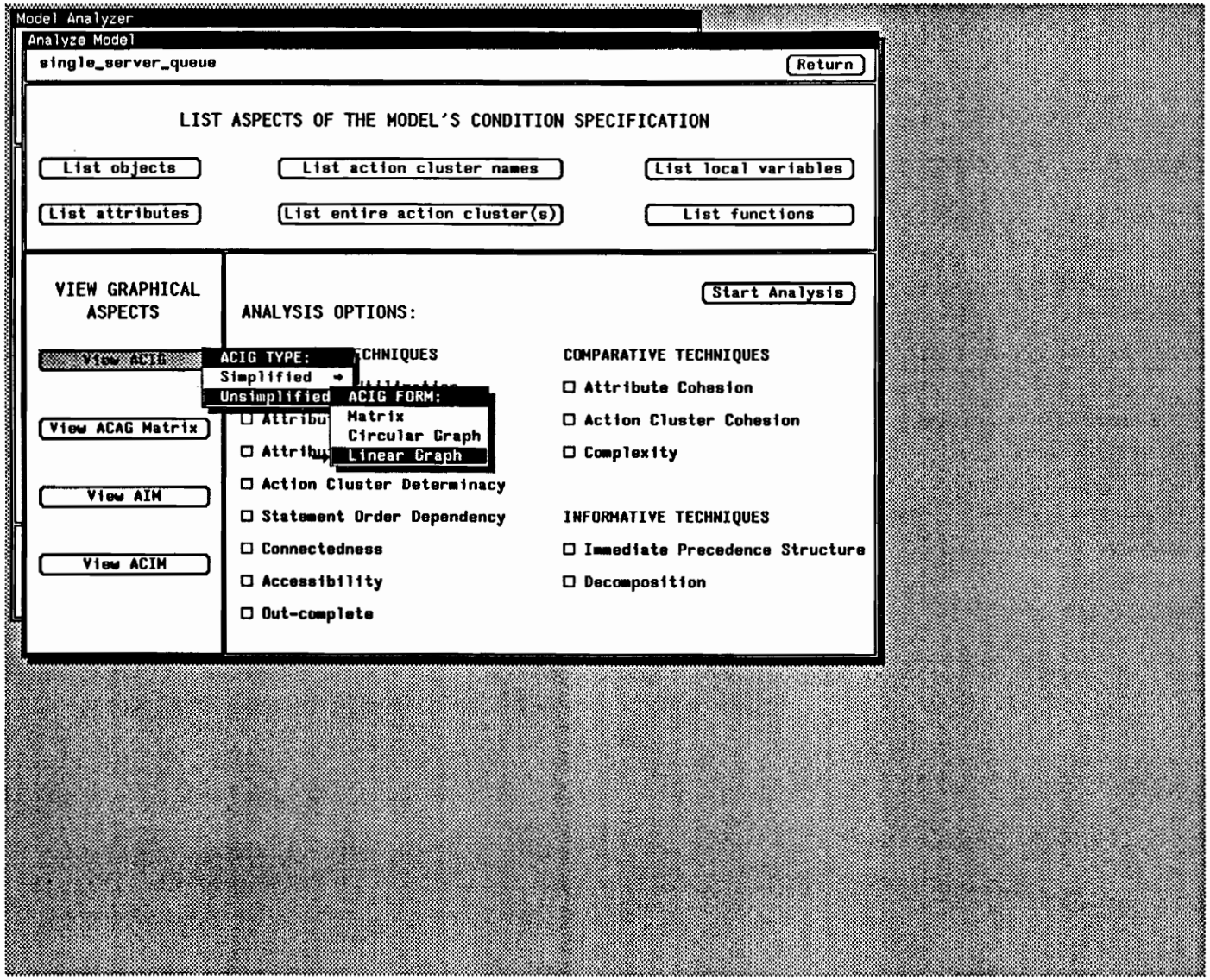


Figure 149. MA: The Series of Menus of the "View ACIG" Button

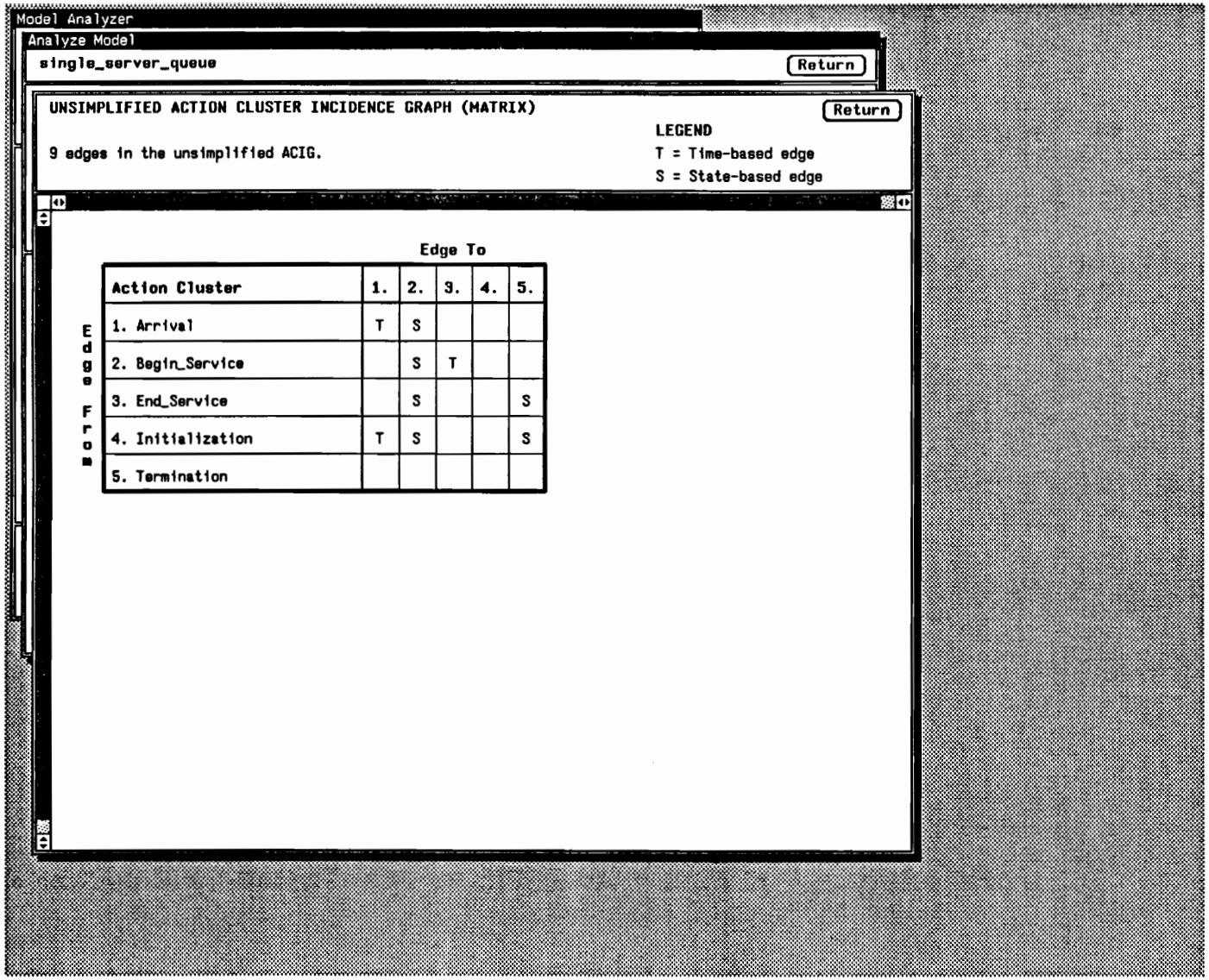


Figure 150. MA: View Matrix Form of Unsimplified Action Cluster Incidence Graph

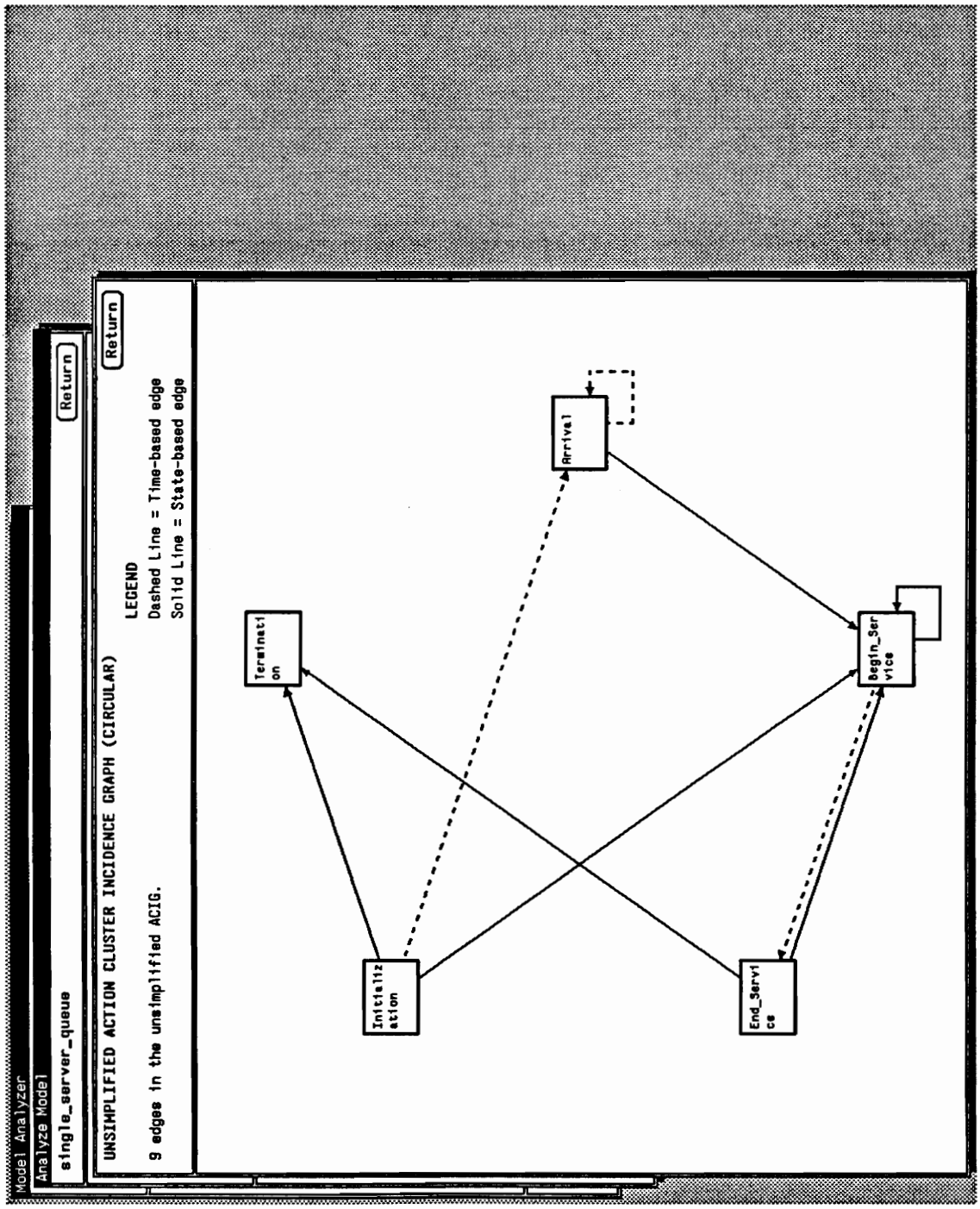


Figure 151. MA: View Circular Form of Unsimplified Action Cluster Incidence Graph

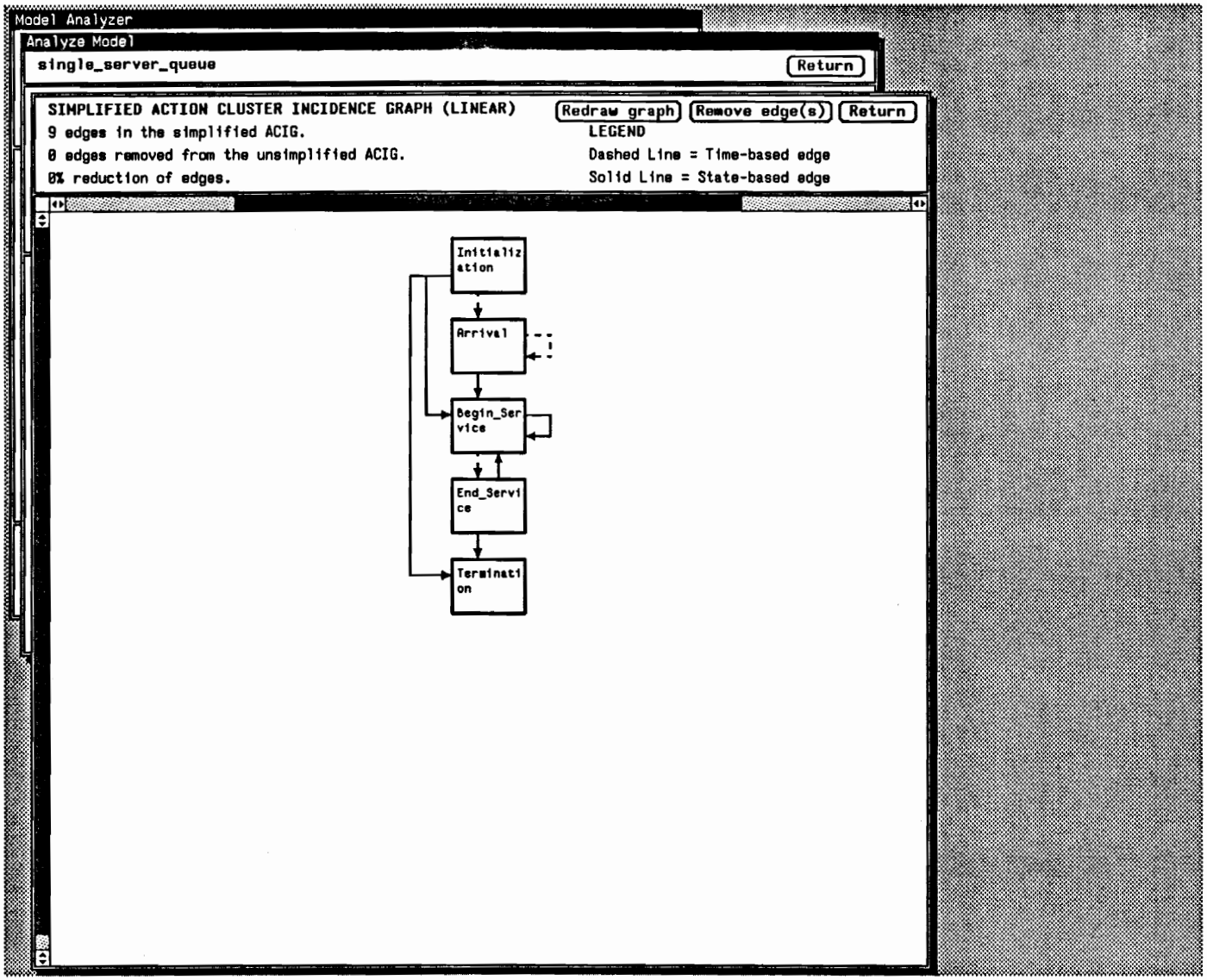


Figure 152. MA: View Linear Form of Simplified Action Cluster Incidence Graph Before Simplification

simplified ACIG is identical with the unsimplified ACIG until the ACIG is simplified by removing infeasible edges from the graph.

H.5.3.1.3 Simplify the Action Cluster Incidence Graph

To simplify the ACIG, view the simplified ACIG as shown above and select the "Remove Edge(s)" button in the Simplified ACIG panel. When the button is selected, the list of all edges in the simplified ACIG appears, shown in Figure 153. Select each infeasible edge one at a time. As each infeasible edge is selected, it disappears from the list. After all infeasible edges are deleted (Figure 154), select the "Redraw Graph" button to view the simplified ACIG without the infeasible edges, shown in Figure 155. Once the simplified ACIG is redrawn, it never again contains any of the removed edges whenever the simplified ACIG is viewed.

H.5.3.2 View Action Cluster Attribute Graph

To view the Action Cluster Attribute Graph (ACAG) of a model, select the "View ACAG Matrix" button. Selecting this button causes the ACAG to be viewed as a matrix, which shows the classification (input, control, output, or time-delayed output) for each edge in the graph, shown in Figure 156.

H.5.3.3 View Attribute Interaction Matrix

To view the Attribute Interaction Matrix (AIM) of a model, select the "View AIM" button. When the button is selected, the AIM is displayed, shown in Figure 157.

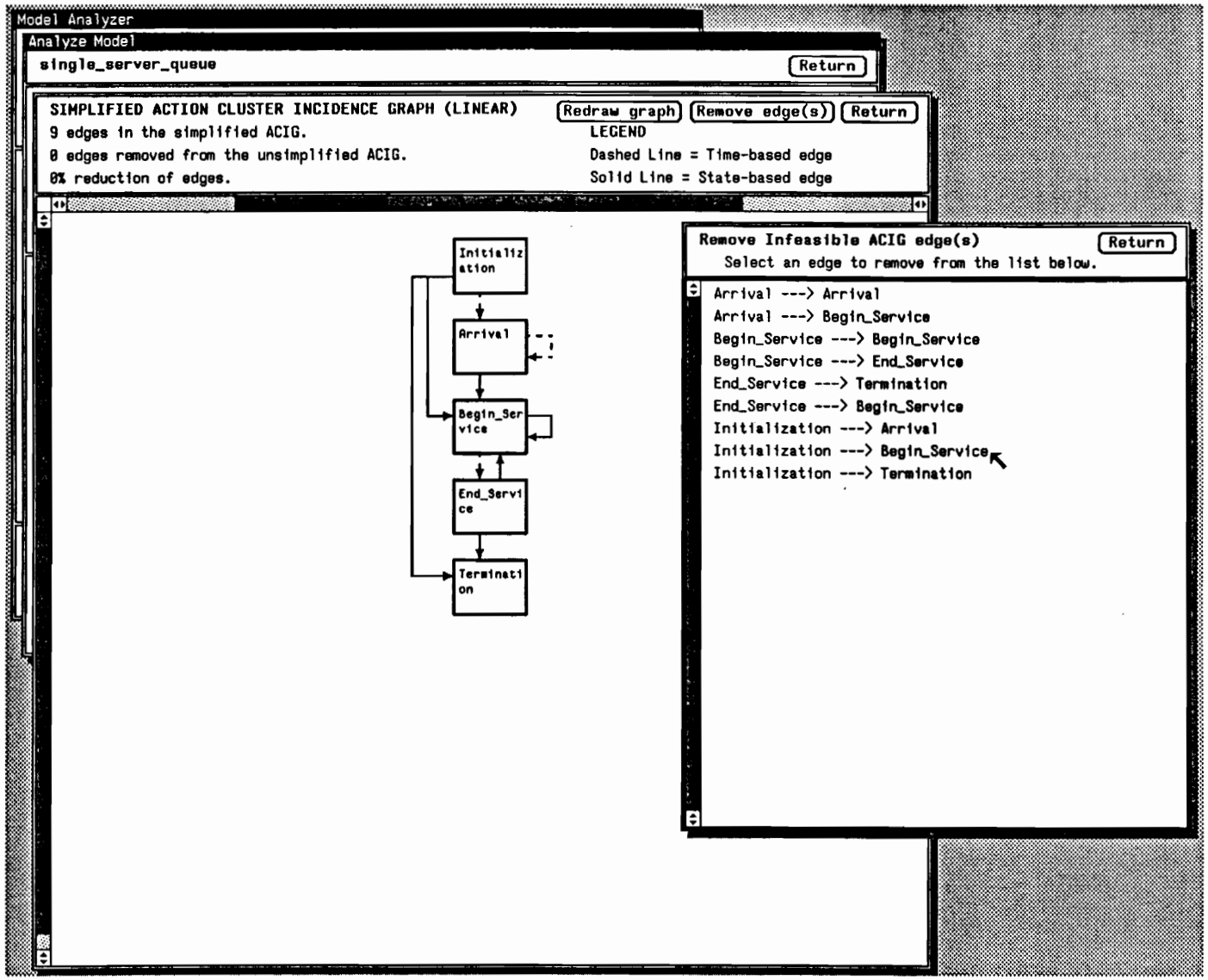


Figure 153. MA: List of Edges in the Simplified Action Cluster Incidence Graph Before Any Infeasible Edges Are Removed

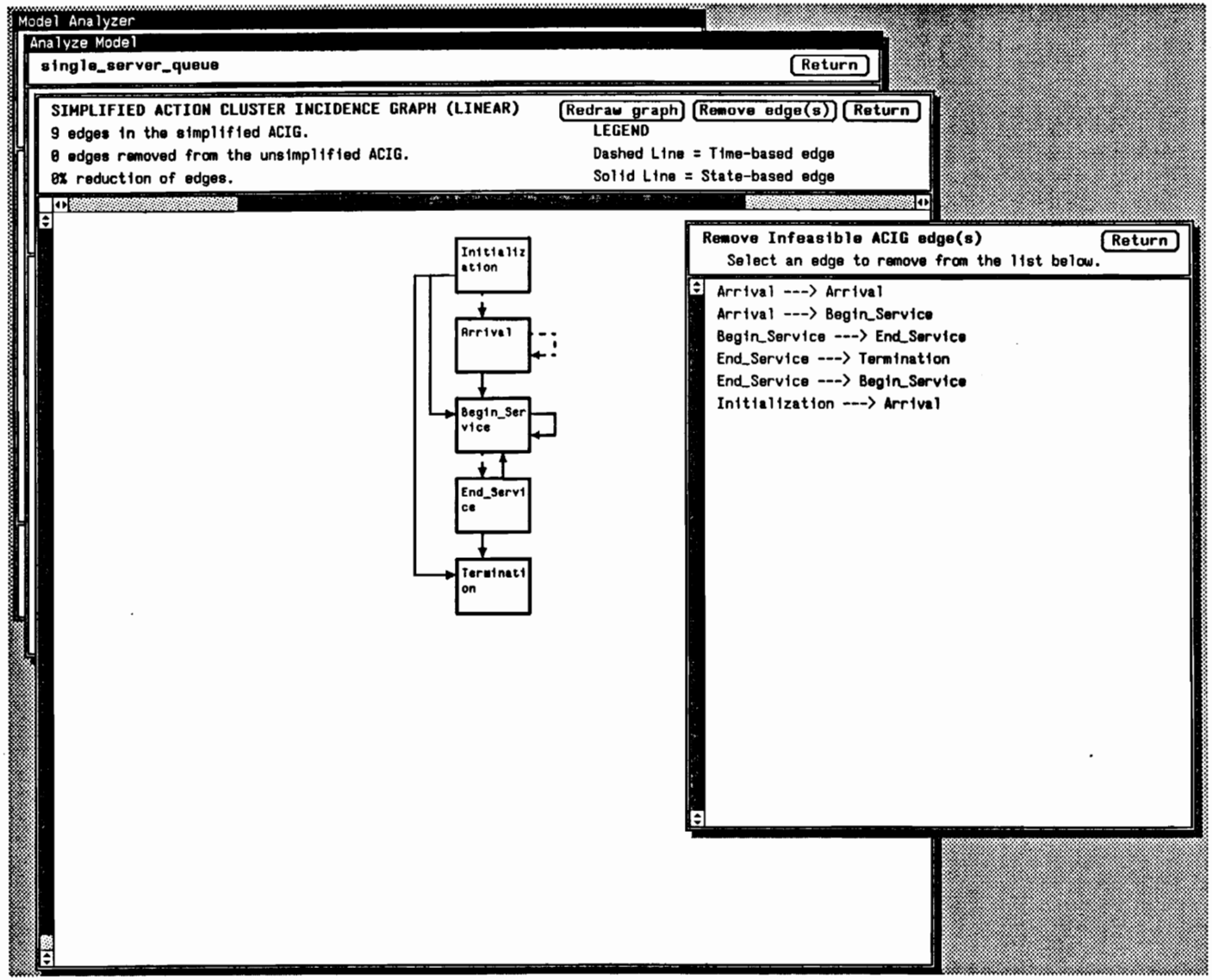


Figure 154. MA: List of Edges in the Simplified Action Cluster Incidence Graph After Three Infeasible Edges Are Removed

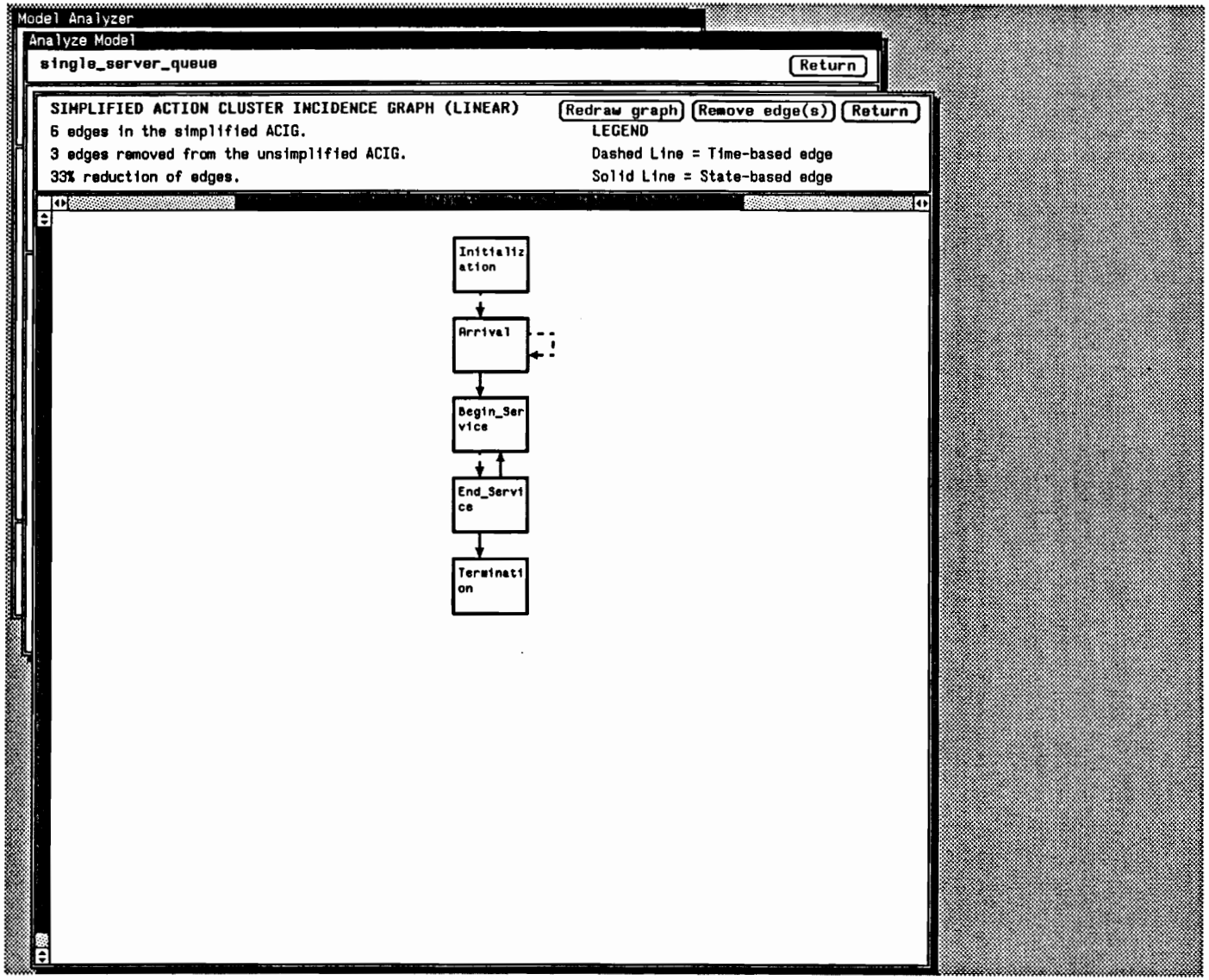
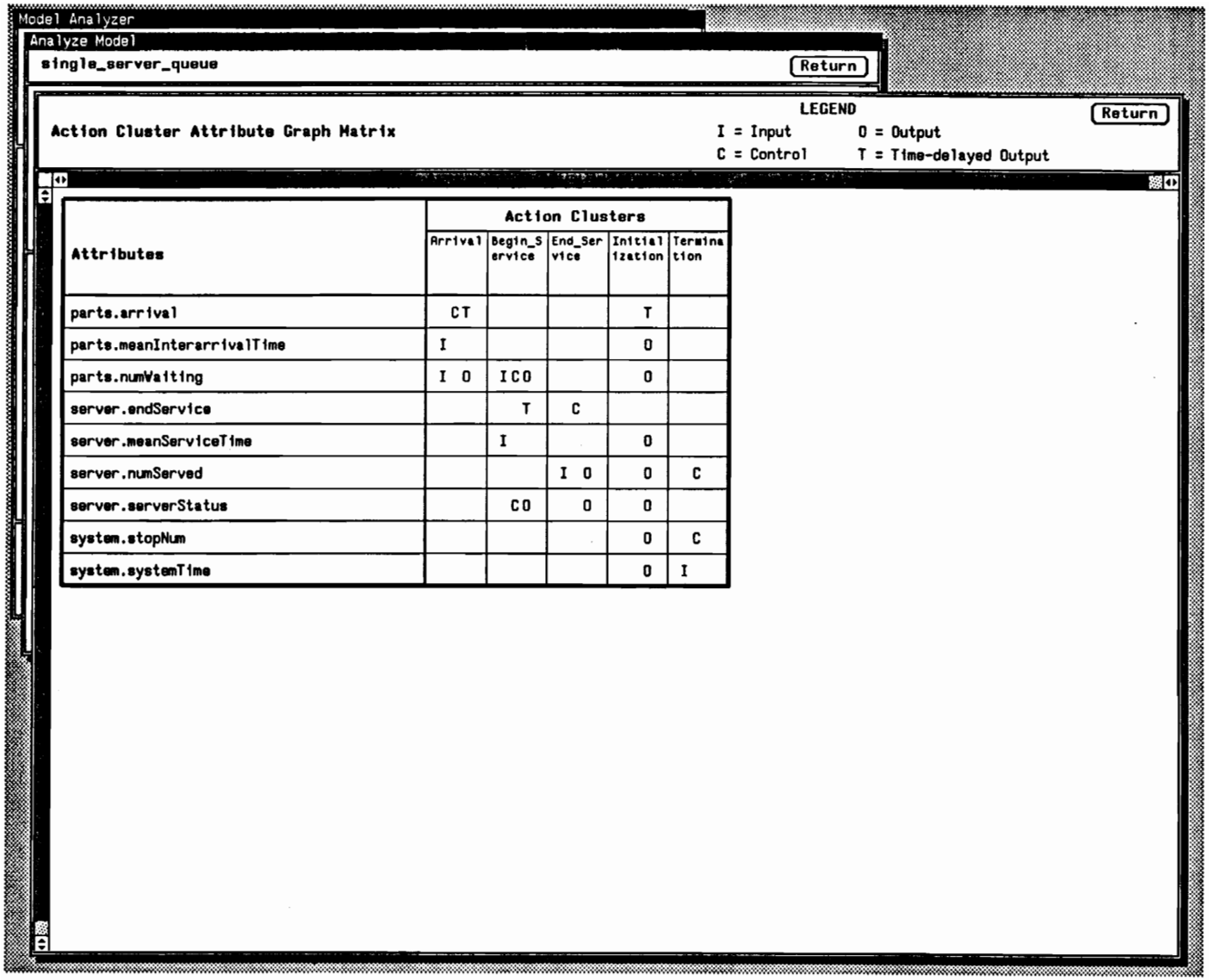


Figure 155. MA: Simplify the Linear Form of the Action Cluster Incidence Graph



Attributes	Action Clusters				
	Arrival	Begin_Service	End_Service	Initialization	Termination
parts.arrival	CT			T	
parts.meanInterarrivalTime	I			O	
parts.numWaiting	I O	ICO		O	
server.endService		T	C		
server.meanServiceTime		I		O	
server.numServed			I O	O	C
server.serverStatus		CO	O	O	
system.stopNum				O	C
system.systemTime				O	I

Figure 156. MA: View the Action Cluster Attribute Graph Matrix

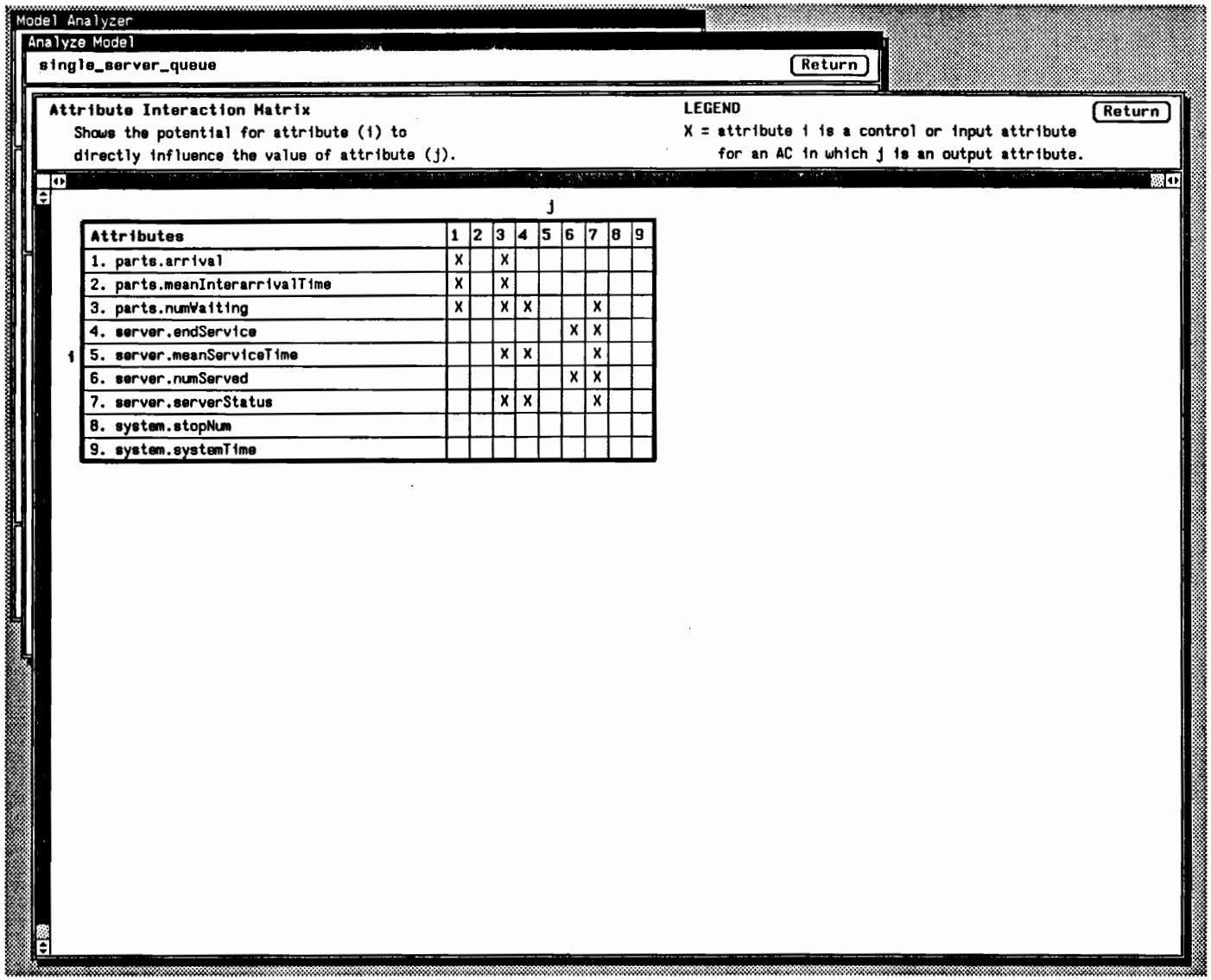


Figure 157. MA: View the Attribute Interaction Matrix

H.5.3.4 View Action Cluster Interaction Matrix

Select the "View ACIM" button to view the Action Cluster Interaction Matrix (ACIM) of a model. When the button is selected, the ACIM, shown in Figure 158, is displayed.

H.5.4 Perform Diagnostic Assistance

The panel in the lower right section of the Model Analyzer frame lists thirteen diagnostic techniques that can be performed upon the model. These techniques, described in the following subsections, fall into three categories: (1) analytical, (2) comparative, (3) and informative. Before performing any of these diagnostic techniques, the ACIG should be simplified to guarantee accurate test results.

To perform diagnostic assistance upon the model, select one or more techniques with the mouse. When a technique is selected, a check mark is displayed in the box next to the name of the technique, shown in Figure 159. A technique is deselected and the check mark removed when a checked technique is selected again. After selecting one or more techniques, depress the "Start Analysis" button to perform the selected techniques.

If the decomposition technique is selected, the modeler chooses the execution order priorities among simultaneously-occurring action clusters before the test results are shown. The execution order priorities are chosen by selecting the action cluster, which has execution priority over the other, or the "Equal Priority" button in a series of alert panels. An example of one alert panel is shown in Figure 160.

The results of the diagnostic techniques are shown in a scrollable text subwindow that appears on the screen, shown in Figure 161. The results for each technique contain a header surrounded by asterisks that states the objective and a brief explanation of the technique. For a more detailed explanation of each diagnostic technique, consult the

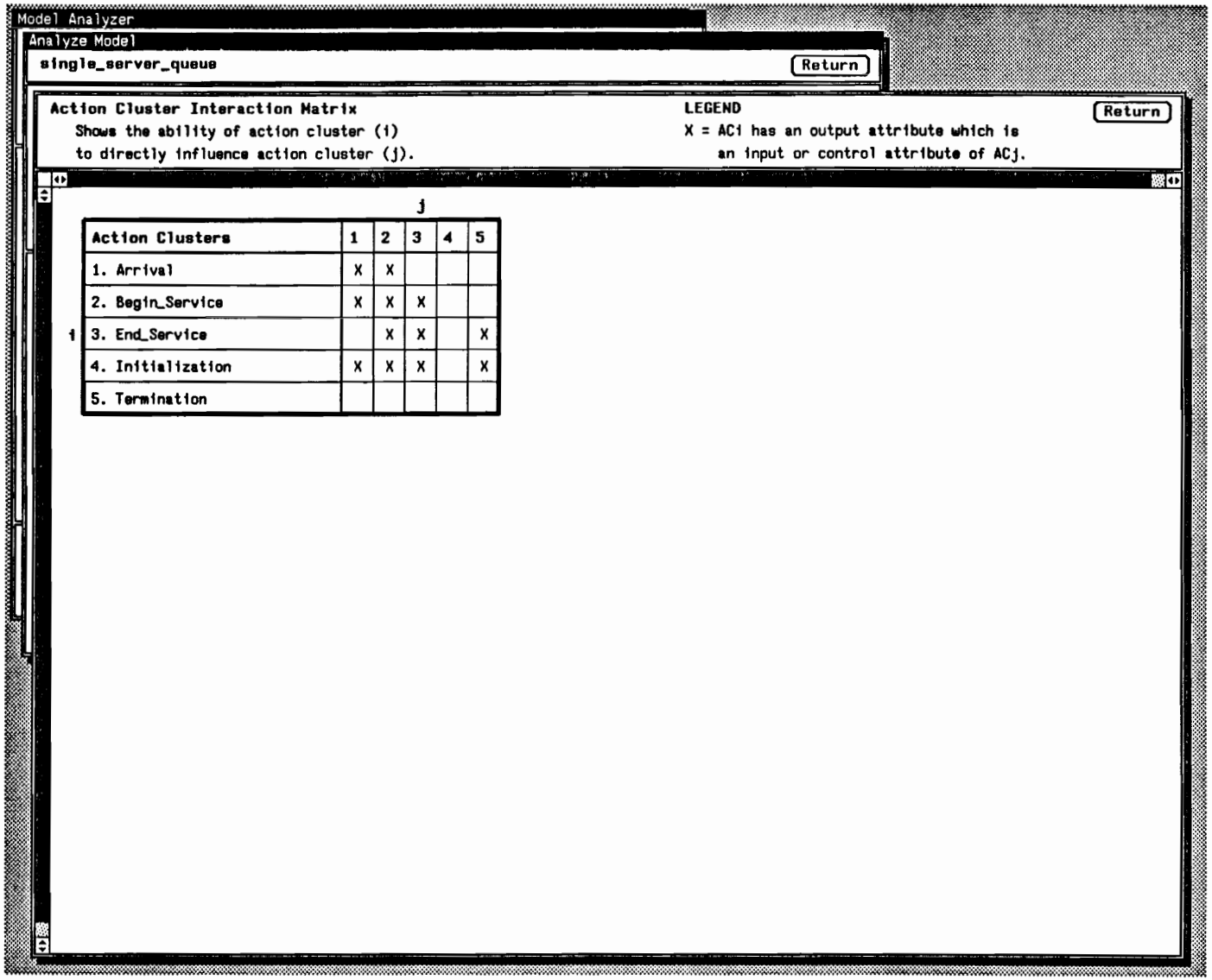


Figure 158. MA: View the Action Cluster Interaction Matrix

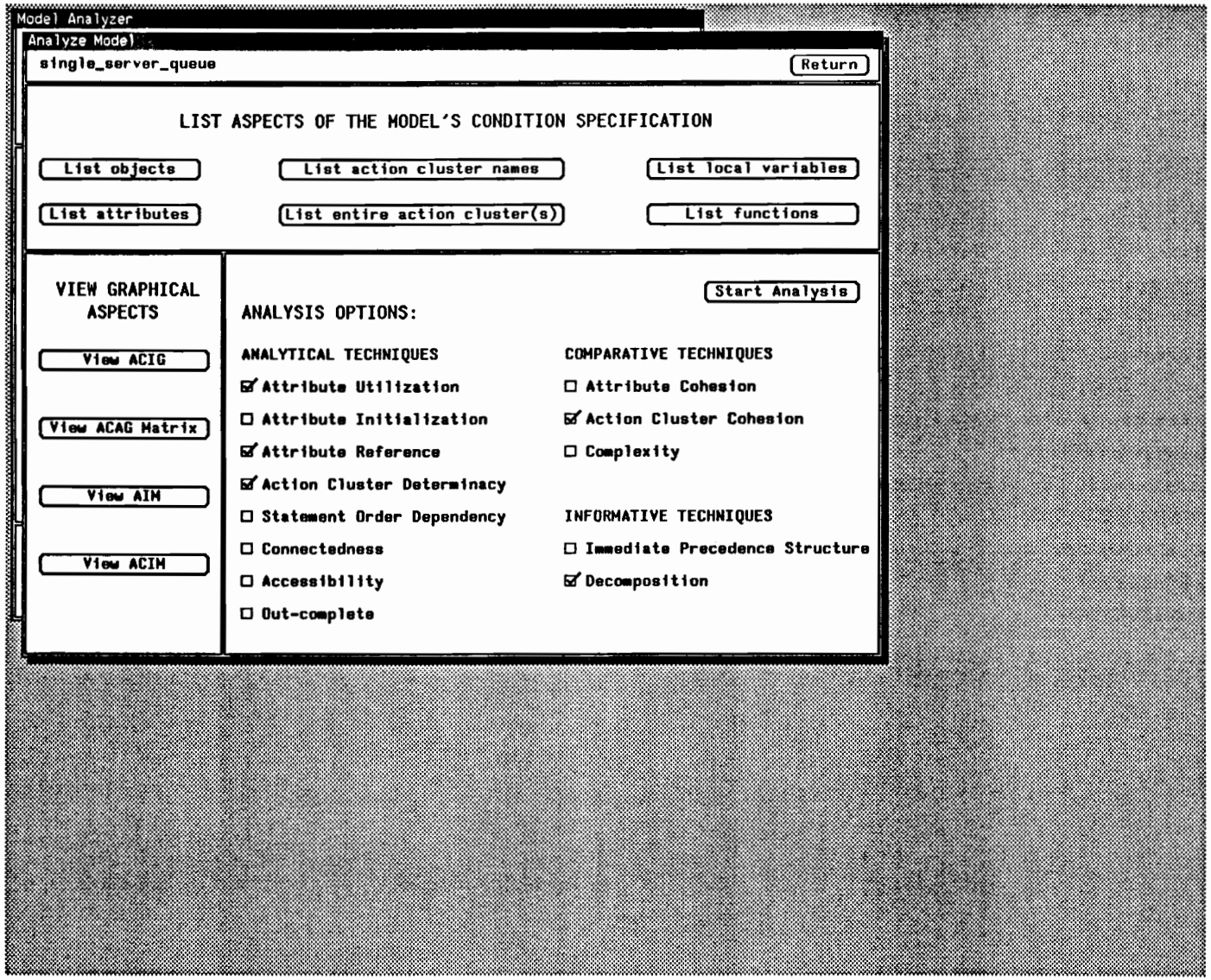


Figure 159. MA: Various Selected Diagnostic Techniques in the Model Analyzer

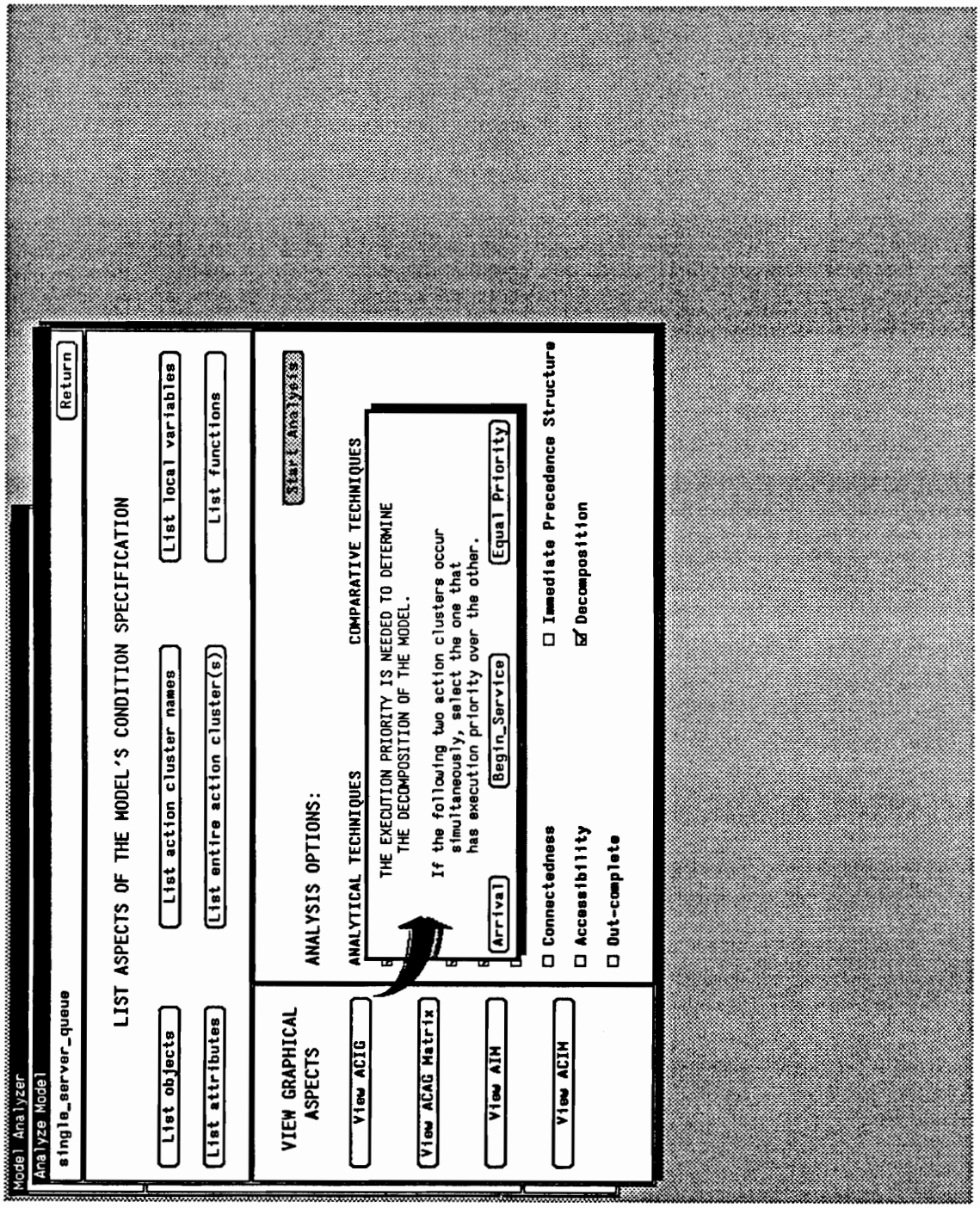


Figure 160. MA: An Alert Panel Used to Determine the Execution Order Priority

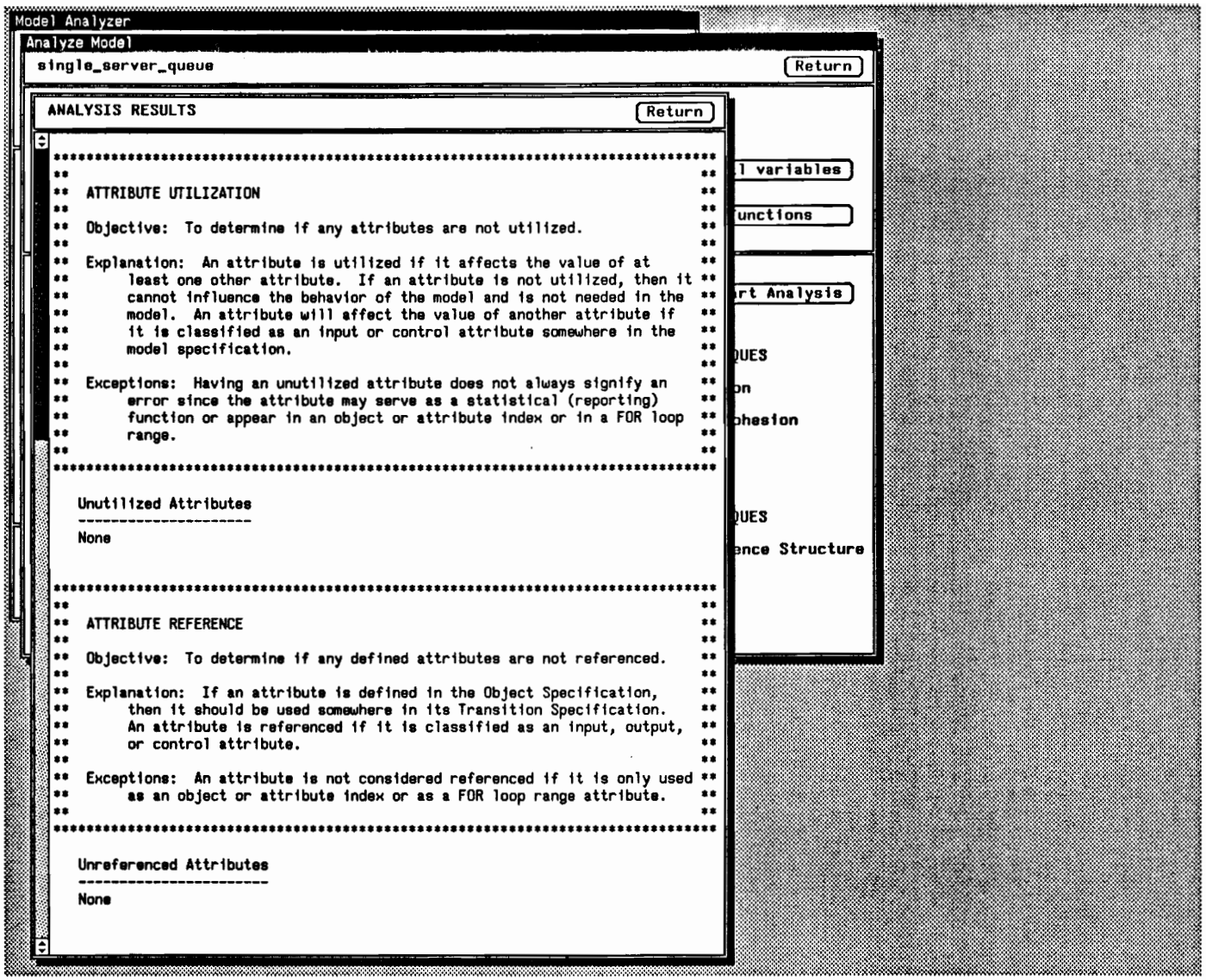


Figure 161. MA: Results of the Selected Diagnostic Techniques

Assistance Manager. For each category of techniques, the objectives for each technique are listed below.

H.5.4.1 Analytical Techniques

Attribute Utilization: To determine if any attributes are not utilized.

Attribute Initialization: To determine if any attributes are not assigned initial values.

Attribute Reference: To determine if any defined attributes are not referenced.

Action Cluster Determinacy: To determine if the required state changes within an action cluster are possible.

Statement Order Dependency: To determine if the ordering of the statements within an action cluster is important.

Connectedness: To determine if the ACIG is connected.

Accessibility: To determine if the ACIG is completely accessible from the Initialization action cluster.

Out-Complete: To determine that no action clusters are specified that cannot influence other action clusters.

H.5.4.2 Comparative Techniques

Attribute Cohesion: To determine the degree to which attribute values are mutually influenced.

Action Cluster Cohesion: To determine the degree to which action clusters are mutually influenced.

Complexity: To determine the complexity of the Condition Specification model.

H.5.4.3 Informative Techniques

Immediate Precedence Structure: To provide information about the data independent (deterministic sequencing of) model actions.

Decomposition: To decompose the ACIG into subgraphs that consist of related action clusters.

Vita

Frederick Anthony Puthoff was born on a small farm in Maria Stein, Ohio, on May 11, 1965. He graduated as class valedictorian from Marion Local High School, Maria Stein, Ohio, in May 1983. In April 1987, he received the degree of Bachelor of Science in Computer Science, Magna Cum Laude, from the University of Dayton in Dayton, Ohio, and was commissioned as an officer in the United States Army after spending four years in ROTC. The U.S. Army sent the author back to school in August 1989, after spending a tour in Korea where he saw the 1988 Olympics in Seoul and met his wife, Hyesuk Kim. In September 1991, he graduated from the Virginia Polytechnic Institute and State University with a Master of Science degree in Computer Science.

Currently, the author is a First Lieutenant on Active Duty in the Signal Corps of the United States Army and is married with no children. He is a member of the honor societies of Upsilon Pi Epsilon and Phi Kappa Phi. He also is a member of the Association for Computing Machinery, Armed Forces Communications and Electronics Association, Association of the United States Army, and the Signal Corps Regimental Association.