MODEL GENERATORS: PROTOTYPING SIMULATION MODEL DEFINITION, SPECIFICATION, AND DOCUMENTATION UNDER THE CONICAL METHODOLOGY

by

Ernest Henry Page, Jr.

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

Deborah Hix

August, 1990

Blacksburg, Virginia

LD
5655
V855
1990
P344
C.2

MODEL GENERATORS: PROTOTYPING SIMULATION MODEL DEFINITION, SPECIFICATION, AND DOCUMENTATION UNDER THE CONICAL METHODOLOGY

by

Ernest Henry Page, Jr.

Richard E. Nance, Chairman

Computer Science

(ABSTRACT)

The process of model generation is key to the realization of a Simulation Model Development Environment. Model generation is facilitated in the environment via the Model Generator - a software utility that assists a modeler in the development of a simulation model specification. Since modeling is inherently creative, the *correct* assistance provided to a modeler can neither be derived algorithmically, nor proved mathematically. Only through experimentation with prototypical assistance forms can we begin to understand the meaning of *correctness*.

This thesis describes the development of a Model Generator prototype for the Simulation Model Development Environment. A review of the literature indicates the need for more extensive questioning of the model generation process to identify the proper foundational support than is available in applications designed under the program generation approach. The Conical Methodology provides the Conceptual Framework, and the Condition Specification provides the target specification form for the Model Generator prototype.

A set of algorithms to derive a condition specification via a series of interactive dialogues is presented, and the results of early prototype experimentation are discussed. New questions are raised as to the role of relational attributes in the Conical Methodology and the extent and types of model analysis provided by a Model Generator. Finally, an analysis of the Model Generator as a platform for the assessment of the Conical Methodology/Condition Specification is given and directions for future research outlined.

Acknowledgements

The author would like to thank the many people who contributed to this work. First and foremost, special thanks to Dick Nance whose hand guided this work, whose funding allowed the author to eat an occasional meal, whose generosity provided the author the opportunity to travel to simulation conferences (and that's not to mention all the blackened redfish he supplied). Thanks to Osman Balci for valuable insights into simulation theory and a hearty congratulations for being one of the few individuals who can beat the author at full-contact racquetball. Thanks to Sean Arthur for his character-building, back-breaking compiler class, and for the power-forward lessons. Thanks to Deborah Hix for her assistance with User Action Notation. A special thanks to the author's wife, Larenda, who has displayed tremendous patience and sacrifice while her husband completed this work -- and maybe a little gullibility since she has agreed to let him pursue a Ph.D. Thanks also to: Larry and Brenda Salyers for the wife and all the steaks; Dr. Lev Malakhoff for demonstrating the availability of higher education to the lesser-minded; Battlin' Bill Baker for teaching the author the game of golf (at the reasonable rate of $1 a hole); Bobby Beamer and Tony Page, without whose company whiling away the hours on the Virginia Tech golf course ... this work would have probably been finished in a more timely manner; Kim Page, for letting Tony play golf with me; Jay Beams, John Bishop, Ed Dorsey, Brian Chappell, John Lewis, Ben Keller, Joel Henry, and Joe Chase for the many intellectual conversations, but especially for their camaraderie

and greatly non-intellectual conversations during the past two years; Joe Derrick for continuing to set the standards by which graduate research is measured, and for oftentimes carrying the burden of the IBM VISION project while the author completed this work; Trish Hubble and Sandra Griffith for their friendship and patience with a learning computer system manager; and thanks to the Los Angeles Lakers for keeping NBA action fantastic during the otherwise indifferent 1980's. Finally, the author wishes to express his undying gratitude to his mother without whose support, intellectual, emotional and financial (and then there was that whole birthing thing), none of the author's life could have taken place ... at any rate, it would have been a great deal less pleasant.

Table of Contents

1 Introduction **1**

1.1 The Importance of the Model Generator 2

1.2 Description of Research 4

2 Literature Review **5**

2.1 Program Generators 5

2.1.1 Application Generators 6

2.1.2 Simulation Program Generators 6

2.2 Specification Languages 10

2.2.1 Current Trends 10

2.2.2 The Future 15

2.3 User Interfaces 15

3 Model Generation **21**

3.1 The Conical Methodology 22

3.1.1 Introduction to the Conical Methodology 23

3.1.2 Model Definition 24

3.1.3 Model Specification 26
3.2 The Condition Specification 27
3.2.1 System Interface Specification 29
3.2.2 Object Specification 29
3.2.3 Transition Specification 29
3.2.4 Report Specification 30
3.2.5 Example 30
3.3 Model Generator Prototypes 35
3.3.1 Box-Hansen 36
3.3.2 Barger 36
3.3.3 Balci-Bishop 36
3.4 Motivation for a New CM-Based Prototype 37
3.4.1 "Dumb-Terminal" Interface Limitations 38
3.4.2 Database Limitations 38
3.4.3 Incomplete Specification 38
3.5 Summary 39

4 Design of a New Model Generator Prototype 40
4.1 Specifying a Model under the CM/CS Framework 41
4.1.1 Indicative Attributes 42
4.1.1.1 Status Transitional 42
4.1.1.2 Temporal Transitional 45
4.1.1.3 Permanent 47
4.1.2 Relational Attributes 47
4.1.2.1 Coordinate 50
4.1.2.2 Hierarchical 52
4.1.3 Time-Based Signals 53
4.1.4 Model Input and Output 54

4.1.5 Object Creation and Destruction 54
4.1.6 Model Initialization and Termination 58
4.1.7 User-Defined Functions 61
4.2 Representing Sets in Model Specification 61
4.3 Specification Completeness 63
4.4 Database Design 66
4.5 Interface Design 67
4.5.1 Model Generator Interface Specification Using User Action Notation 68
4.5.2 Exploitation of UNIX Multitasking 69
4.5.3 Advantages of Iconic Interfaces 72
4.5.4 Explicit Display of Model Hierarchy 72
4.5.5 Rapid Shift Between Definition and Specification 74
4.5.6 Automatic Revision for Configuration Management 74
4.6 Summary 76

5 Example 77
5.1 Initial Step in Model Definition 80
5.2 Defining Model Objects 80
5.3 Model Specification 84
5.3.1 Specifying Status Transitional Indicative Attributes 90
5.3.2 Specifying Alarms 99

6 Evaluation of the Model Generator 110

7 Summary and Conclusions 115
7.1 Future Research 116
7.1.1 Prototype Improvements 116
7.1.2 A New Direction 118

7.2 Conclusions 119

Bibliography **121**

Appendix A. Model Generator Interface Specification Diagrams in User Action Notation . **131**

Appendix B. Model Generator User's Manual **151**

B.1 Create a Model 152

B.2 Retrieve a Model 152

B.3 Exit the Model Generator 153

B.4 Define the Model 153

B.4.1 Attach Attributes 153

B.4.2 List Attributes 154

B.4.3 Make a Subobject 154

B.4.4 Delete the Object 155

B.4.5 Retrieve a Subobject 155

B.4.6 Create a Set 155

B.4.6 Modify Model Definition 156

B.5 Specify the Model 156

B.5.1 Attributes 157

B.5.1.1 Status Transitional Indicative 158

B.5.1.2 Temporal Transitional Indicative 158

B.5.1.3 Permanent Indicative 159

B.5.1.4 Relational 159

B.5.1.5 Time-Based Signal 160

B.5.2 Conditions 160

B.5.2.1 State-Based 161

B.5.2.2 Time-Based 161

B.5.2.3 Mixed 162
B.5.3 Object Creation and Destruction 162
B.5.4 Set Membership for an Object 162
B.5.5 Initialization 162
B.5.5.1 System Time 163
B.5.5.2 P-sets 163
B.5.6 Termination 164
B.5.7 Model Input and Output 164
B.5.8 Functions 165
B.5.9 Monitored Routines 165
B.5.10 Modify Model Specification 165
B.6 List and Edit Features 165

Vita 211

List of Illustrations

Figure 1. The Architecture of the SMDE 3
Figure 2. The Simulation Model Life-Cycle 11
Figure 3. User Engineering Principles for Interactive Systems 18
Figure 4. The Design of Idiot-Proof Interactive Programs 19
Figure 5. Design Guidelines for Interactive Systems 20
Figure 6. The Tree of Conical Methodology Types 25
Figure 7. Machine Repairman System Interface Specification 32
Figure 8. Machine Repairman Object Specification 33
Figure 9. Machine Repairman Transition Specification 34
Figure 10. Barger's Technique for Status Attribute Specification 43
Figure 11. Algorithm for Status Transitional Indicative Attribute Specification 44
Figure 12. Algorithm for Temporal Transitional Indicative Attribute Specification 46
Figure 13. Algorithm for Permanent Indicative Attribute Specification 48
Figure 14. Algorithm for Alarm Specification 49
Figure 15. Algorithm for Relational Attribute Specification 51
Figure 16. Algorithm for Time-Based Signal Specification 55
Figure 17. Algorithm for Model Input and Output Specification 56
Figure 18. Algorithm for Object Creation and Destruction Specification 57
Figure 19. Algorithm for Initialization Specification 59
Figure 20. Algorithm for Termination Specification 60
Figure 21. Algorithm for Function Specification 62

Figure 22. Algorithm for Set Specification 65

Figure 23. UAN Example -- Deleting a Macintosh File 71

Figure 24. The Model Generator in the Simulation Model Development Environment. 73

Figure 25. The Model Hierarchy Display. 75

Figure 26. Model Generator Produced Machine Repairman Object Specification 78

Figure 27. Model Generator Produced Machine Repairman Transition Specification 79

Figure 28. Entering Model Definitions. 81

Figure 29. The Definition/Specification Driver. 82

Figure 30. Editing Model Documentation. 83

Figure 31. Model Definition Menu. 85

Figure 32. Entering New Model Object. 86

Figure 33. New Object Becomes Current Object. 87

Figure 34. Attaching Object Attributes. 88

Figure 35. Listing Object Attributes. 89

Figure 36. Model Specification Menu. 91

Figure 37. Object Specification Menu. 92

Figure 38. Selecting Attribute to Specify. 93

Figure 39. The Modeler Must Provide a CS Type for Attributes 94

Figure 40. Enumerating the Values of Status Transitional Indicative Attributes. 95

Figure 41. The Status Transitional Indicative Attribute Specification Menu. 96

Figure 42. Adding Value Changes for Status Transitional Indicative Attributes. 97

Figure 43. Listing Value Changes for Status Transitional Indicative Attributes. 98

Figure 44. Specifying Conditions for Change in Value of a Status Transitional Indicative Attribute. 100

Figure 45. Supply Condition Menu. 101

Figure 46. Specifying a New Condition. 102

Figure 47. Entering a Boolean Expression for a State-based Condition. 103

Figure 48. The Completed CAP Display. 104

Figure 49. Listing Conditions Causing Value Change for a Status Transitional Indicative Attribute. 105
Figure 50. Specifying the Condition Arrive_idle. 106
Figure 51. Specifying an Alarm for a Time-based Condition. 107
Figure 52. The Completed CAP Display for Arrive_idle. 108
Figure 53. Alarms are Attributes of the Top Level Object. 109
Figure 54. UAN : Select_Object 132
Figure 55. UAN : Attach_Attributes 133
Figure 56. UAN : Make_Subobject 134
Figure 57. UAN : List_Attributes 135
Figure 58. UAN : Get_Condition 136
Figure 59. UAN : Select_Existing_Condition 137
Figure 60. UAN : Select_New_Condition 138
Figure 61. UAN : Add_Mixed_Condition 139
Figure 62. UAN : Add_State_Condition 140
Figure 63. UAN : Add_Time_Condition 141
Figure 64. UAN : Add_New_Alarm 142
Figure 65. UAN : Select_Existing_Alarm 143
Figure 66. UAN : Specify_Sti_Attribute 144
Figure 67. UAN : Add_Value_Change 145
Figure 68. UAN : Delete_Value_Change 146
Figure 69. UAN : List_Value_Change 147
Figure 70. UAN : Spec_Sti_Caps 148
Figure 71. UAN : Get_Value_Change 149
Figure 72. UAN : Function Definitions 150
Figure 73. MG : The Model Generator in the Simulation Model Development Environment. 167
Figure 74. MG : Entering Model Definitions. 168
Figure 75. MG : Retrieving Existing Models. 169
Figure 76. MG : The Definition/Specification Driver. 170

Figure 77. MG : Model Definition Menu. 171

Figure 78. MG : Attaching Object Attributes. 172

Figure 79. MG : Listing Object Attributes. 173

Figure 80. MG : Entering New Model Object. 174

Figure 81. MG : New Object Becomes Current Object. 175

Figure 82. MG : Entering Set Information. 176

Figure 83. MG : Set Definition Menu. 177

Figure 84. MG : Modify Object Menu. 178

Figure 85. MG : Modifying Attribute Definition. 179

Figure 86. MG : Model Specification Menu. 180

Figure 87. MG : Object Specification Menu. 181

Figure 88. MG : Selecting Attribute to Specify. 182

Figure 89. MG : Enumerating the Values of Status Transitional Indicative Attributes. 183

Figure 90. MG : The Status Transitional Indicative Attribute Specification Menu. 184

Figure 91. MG : Adding Value Changes for Status Transitional Indicative Attributes. 185

Figure 92. MG : Listing Value Changes for Status Transitional Indicative Attributes. 186

Figure 93. MG : Specifying Conditions for Change in Value for Status Transitional Indicative Attributes. 187

Figure 94. MG : Listing Conditions Causing Value Change for Status Transitional Indicative Attributes. 188

Figure 95. MG : Non-status Attribute Specification Menu. 189

Figure 96. MG : Action Menu for Temporal and Permanent Attributes. 190

Figure 97. MG : Supply Condition Menu for Permanent Indicative Attributes. 191

Figure 98. MG : Action Menu for Relational Attributes. 192

Figure 99. MG : Action Menu for Time-based Signal Attributes. 193

Figure 100. MG : Supply Condition Menu. 194

Figure 101. MG : Selecting an Existing Model Condition. 195

Figure 102. MG : Specifying a New Model Condition. 196

Figure 103. MG : The Completed CAP Display. 197

Figure 104. MG : Entering a Boolean Expression for a State-based Condition. 198
Figure 105. MG : Specifying an Alarm for a Time-based Condition. 199
Figure 106. MG : Supply Parameter Menu for a Time-based or Mixed Condition. 200
Figure 107. MG : Supply Identifier Menu for Object Creation and Destruction. 201
Figure 108. MG : Initialization Specification Menu. 202
Figure 109. MG : Providing an Initial Value for System Time. 203
Figure 110. MG : Specifying Distinguishing Attribute for P-sets. 204
Figure 111. MG : Termination Specification Menu. 205
Figure 112. MG : Providing an Expression for Termination. 206
Figure 113. MG : Specify Input/Output Menu. 207
Figure 114. MG : Listing Action Clusters. 208
Figure 115. MG : Editing Model Documentation. 209
Figure 116. MG : Editing a Model List Item. 210

List of Tables

Table 1. A Sample of Production Simulation Program Generators 9

Table 2. The Phases of a Generic Software Life-Cycle Model 13

Table 3. A Taxonomy of Specification Languages 14

Table 4. Syntax and Function of Condition Specification Primitives 28

Table 5. Syntax and Function of Set Operations 64

Table 6. Summary of UAN Notation 70

Table 7. Specification Properties 111

List of Acronyms

CAP	--	Condition Action Pair
CF	--	Conceptual Framework
CM	--	Conical Methodology
CS	--	Condition Specification
DBMS	--	Database Management System
ER	--	Entity-Relationship
GPVSS	--	General Purpose Visual Simulation System
MG	--	Model Generator
MMDE	--	Minimal Model Development Environment
PBQ	--	Programming By Questionnaire
SMDE	--	Simulation Model Development Environment
SMSDL	--	Simulation Model Specification and Documentation Language
SPL	--	Simulation Programming Language
STM	--	Short Term Memory
UAN	--	User Action Notation
pi	--	permanent indicative
sti	--	status transitional indicative
tti	--	temporal transitional indicative

1 Introduction

Computer simulation is a valuable problem solving technique. When properly applied, it offers significant benefits over analytical techniques [Fishman 1973; Shannon 1975]. Unfortunately, the value of simulation is tempered by the high cost of developing and maintaining simulation models. Roth, *et al.* [1978] identify the United States Government as the largest sponsor and consumer of models in the world. Annual expenditures on modeling related issues are estimated to exceed one-half billion dollars [Hansen 1984]. As societal pressures force the Government to become more debt-conscious, a technology which provides for cost-effective simulation modeling efforts *must* be developed if simulation is to remain a viable solution technique for both industrial and military applications in the next century. One effort to reduce the cost of simulation model development and maintenance is the Simulation Model Development Environment (SMDE) [Balci 1986a; Balci and Nance 1987a]. The SMDE is intended to provide automated support throughout the simulation model life-cycle.

The research described in this thesis focuses on discrete-event systems simulation, more specifically, a principal component of the SMDE, the Model Generator: a software utility that assists a modeler in the creation of model representations following a process governed by the Conical Methodology [Nance 1981a, 1987].

1.1 The Importance of the Model Generator

The goal of the SMDE is to provide an *integrated* set of software utilities that offer automated support in the development, analysis, translation, verification, and archival storage and retrieval of discrete event simulation models. The SMDE project seeks to achieve the automation-based paradigm [Balzer et al. 1983] through the evolutionary development of prototypes [Balci and Nance 1987a]. The architecture of the SMDE is pictured in Figure 1. This support system has the potential for significant reduction of the cost of developing and using simulation models.

Balci [1986a] identifies the Model Generator (MG) as one of the essential components of a Minimal Model Development Environment (MMDE). The requirements for the MG are to:

(1) Create a specification of a model in a predetermined analyzable form which is independent of any Simulation Programming Language (SPL) (or any other solution technique oriented representation).

(2) Create multilevel (stratified) model documentation.

(3) Effectively assist in model qualification.

The specification generated by the MG must be completely translatable into executable code and application domain independent. The effectiveness with which the MG can achieve its requirements relates directly to the power of the Conceptual Framework (CF) which underlies the tool. Current research indicates that no existing CF in-and-of-itself provides an ideal CF on which to base the Model Generator [Derrick 1988]. Until the ideal CF can be found, or in case *no* such CF is possible, research directed toward realizing the greatest utility from existing CFs is essential.

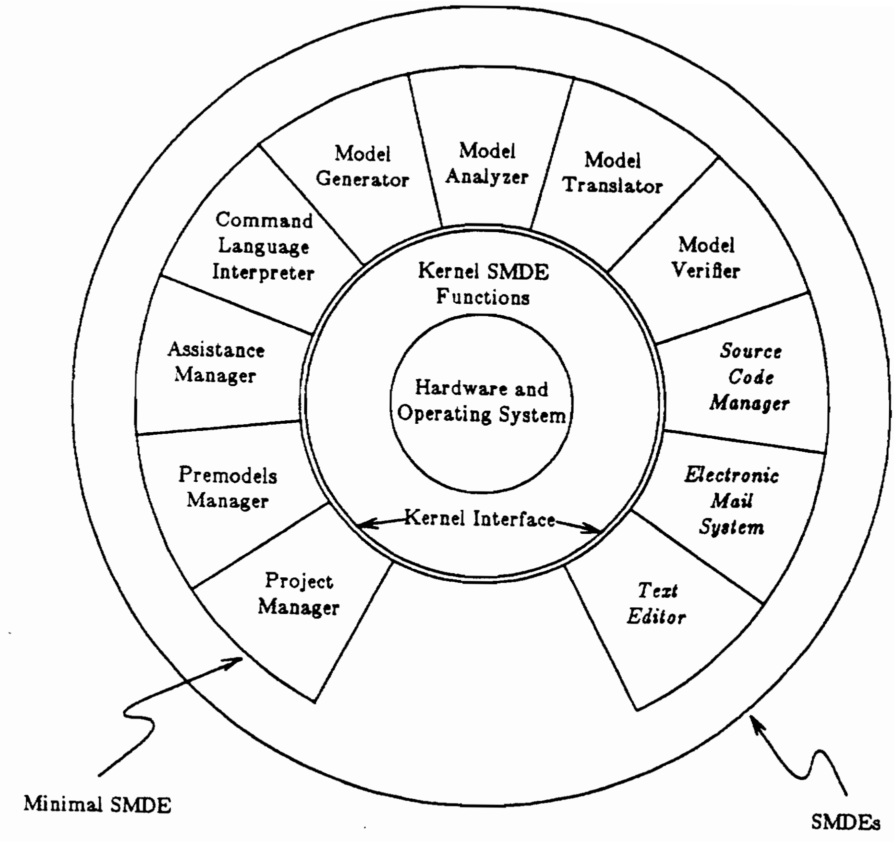

Figure 1. The Architecture of the SMDE
[Balci 1986a]

1.2 Description of Research

The Conical Methodology (CM) [Nance 1981a, 1987] offers a CF that, although less than ideal, provides significant support in the design of a Model Generator. This research aims to provide a platform for analyzing the potential of the CM as the CF for a Model Generator. The research seeks to achieve its goals through the development of, and experimentation with, a CM-based Model Generator prototype that significantly advances the capabilities of previous prototyping efforts and meets the stated requirements for the Model Generator in the SMDE.

Chapter 2 reviews the relevant literature in the areas of program generation, specification languages, and user interfaces. Model generation, the CM, and previous Model Generator prototypes are discussed in Chapter 3. Chapter 4 describes in detail the design of the new prototype. A brief example is presented in Chapter 5. In Chapter 6, an evaluation of the prototype is offered. Conclusions and directions for future research are outlined in Chapter 7.

2 Literature Review

An understanding of the requirements for a successful Model Generator prototype necessitates a review of the current literature in a variety of areas. Section 2.1 deals with program generation approaches. Section 2.2 surveys specification languages, and user interface strategies are reviewed in Section 2.3. (Previous Model Generator prototypes are reviewed in Chapter 3.)

2.1 Program Generators

A program generator is a program that *produces* another program in a high-level language from a simple input description [Clementson 1973]. In essence, the "simple input description" is a user's specification of the target program. The form the specification takes varies widely according to the application. The first program generator was the Programming by Questionnaire (PBQ) system developed at RAND [Oldfather et al. 1966, 1967] which generated Job Shop simulations based on information provided by a modeler via machine-readable responses to a questionnaire.

2.1.1 Application Generators

To date, the prevalent work in program generation has been in the area of application generators. Application generators can be characterized as generating programs for typically data-intensive applications. In most cases, the application relies heavily on a Database Management System (DBMS). Many application generators also function as report generators; these utilities extract information from a database and format it into a report. Martin [1982] provides a detailed analysis of production application generators, as well as the theory underlying the application generator approach. A review of the methods for application generation, such as querying techniques and Fourth-Generation languages, shows them to be of minimal use in the area of simulation program generation, therefore a more detailed treatment here is inappropriate.

2.1.2 Simulation Program Generators

A simulation program generator is an interactive software tool that translates the logic of a model described in a relatively general symbolism into the code of a simulation language and so enables a computer to mimic model behavior [Mathewson 1984]. According to Mathewson [1975], the steps in the use of a simulation program generator are to:

(1) Prepare a symbolic description of the model.

(2) Use the program generator to obtain a translation of the symbolic description into a simulation program.

(3) Edit the simulation program to insert further detail whose representation is outside the scope of the symbolic logic.

Most of the early work in simulation program generators was conducted in Europe and based on the entity cycle (or activity cycle) diagram. Originally labeled "wheel charts" by [Tocher 1964], details of entitiy cycle diagrams can be found in [Hills and Poole 1969]. These early generators, CAPS/ECSL [Clementson 1973], DRAFT [Mathewson 1974], and GASSNOL [Vidallon 1980] a simulation program generator based on the Network Oriented CAD Input Language (NOCADIL), are application specific and reflect a single world-view (the activity scan or "three-phase approach"). Of the British work, DRAFT may be the most widely exercised. It has been adapted to generate programs in FORTRAN, GASP, and SIMSCRIPT and more recently extended to provide model animation on an IBM PC [Mathewson 1978, 1984].

Recently, however, the focus of the simulation community has shifted from a program view of the simulation process to a model view [Nance 1983]. This shift in focus has resulted in more research into modeling *environments* in which simulation programs can be developed, or translated automatically, from a model specification, and less attention to issues in *pure* simulation program generation. The environment approach eliminates the need for step (3) identified above. The SMDE espouses this philosophy. Other research in the environment domain include, CASM [Balmer and Paul 1986; Paul and Chew 1987], KBS [Reddy et al. 1986], JADE [Unger et al. 1986], ANDES [Birtwistle et al. 1984], and TESS [Standridge et al. 1985]. For an overview of these efforts, as well as an analysis of the requirements of simulation environments, refer to [Balci 1986a; Balci and Nance 1987b].

Commercially, several simulation program generators are currently available. Rohrbough [1989] describes a CACI product, SIMFACTORY II.5, that produces SIMSCRIPT II.5 simulations of manufacturing systems without programming. A mouse-driven graphical user interface enables a modeler to construct a graphical representation of a system by manipulating graphics primitives provided in the package. To simulate a system a modeler first graphically defines a factory, then the products produced by the factory, and finally specifies run options such as run length, number of replications, etc. A similar CACI product, NETWORK II.5, provides program generation for networks of queues [Garrison 1989].

Many current simulation program generators provide animation of the model, generally post-simulation animation. In some, such as AutoMod II, the animation is the primary focus of the application. Designed for use on Silicon Graphics workstations, AutoMod II is a CAD-based generator for manufacturing systems. Lenz [1989] describes MAST, which provides both data-driven and graphically-oriented descriptions for manufacturing systems. A summary of some of these products appears in Table 1.

As with the early simulation program generators, current generators are limited in their domain and most suffer from a lack of a sound modeling methodology. They have been constructed as "overlays" on existing software without extensive questioning to identify the proper foundational support. Worse yet, many suffer from overinflated egos; according to [Thompson 1989], "AutoMod II is the ULTIMATE simulation system builder!"

The greatest criticism of the program generation approach may be that following this approach, in the long run, could result in a "Tower of Babel" in the simulation community. The rise of the many network and manufacturing dialects is a natural response of manufacturer's marketing SPLs and related services; the desire to "give the customer what he wants (and nothing more)," and to deliver a product to market in a short period naturally leads to the development of highly domain specific utilities. However, this "quick-fix" method leads to an over abundance of similar products. Nance [1989] observes:

> Already, we find an extension of the network dialect to particular types of networks -- COMNET II.5 for telecommunications networks. Does the future portend dialects for transportation networks, petroleum pipeline networks, and retail distribution networks? Where does the perceived cost to accommodate user abstraction give way to the restrictions of hard-coded interpretation and the ongoing cost of learning and maintaining *one more dialectal tool?*

The questions above promote the search for modeling assistance that: (1) adopts to the user's need rather than anticipating it, and (2) recognizes and utilizes general principles and concepts rather than repackaging them.

Table 1. A Sample of Production Simulation Program Generators

PRODUCT	MODEL	DOMAIN	ANIMATION	REFERENCES
AutoMod II	Graphical	Manufacturing	Yes	[Thompson 1989]
CAPS/ECSL	Entity cycle diagram	Queueing Networks	No	[Clementson 1973]
DRAFT	Entity cycle diagram	Queueing Networks	Yes	[Mathewson 1974] [Mathewson 1975] [Mathewson 1984]
GASSNOL	NOCADIL	Queueing Networks	No	[Vidallon 1980]
MAST	Graphical Data-driven	Manufacturing	Yes	[Lenz 1989]
NETWORK II.5	Graphical	Queueing Networks	Yes	[Garrison 1989]
SIMFACTORY II.5	Graphical	Manufacturing	Yes	[Rohrbough 1989]

2.2 Specification Languages

Specification is the process of describing system behavior in order to assist the system designer in converting the model in his/her mind (conceptual model) into a model that can be communicated to others (communicative model). The purpose of specification in the development of software is to separate "WHAT" the system is to do from "HOW" it is to be done [Balzer and Goldman 1979].

Within the simulation domain, specification plays an important role: it is crucial to the communicative models phase of the simulation model life-cycle as defined by Nance [1981a] and Balci [1986b] and illustrated in Figure 2. The detection of errors as early as possible in the life-cycle is critical in large modeling efforts. Errors induced within the specification of the communicative model, if caught in a much later phase, result in a higher cost of correction. Worse yet, errors never detected can lead to type II error -- the error of accepting invalid results [Balci 1986b].

2.2.1 Current Trends

A specification language offers perceptual guidance by the provision of concepts describing behavior [of a system]; but, more importantly, a specification language is the medium of communication for expressing this behavior [Overstreet et al. 1986]. Many successful specification languages have been developed within the software engineering community. An informal review of a variety of these languages is provided by way of a comparison matrix in [Stoegerer 1984]. A detailed treatment of software specification languages can be found in [Barger 1986; Overstreet et al. 1986].

Software specification languages can be categorized according to the phase of a generic software life-cycle model supported. The phases of such a generic model are illustrated in Table 2. Balzer et al. [1983] and Zave [1984a] propose an alternate view of the traditional life-cycle known as the

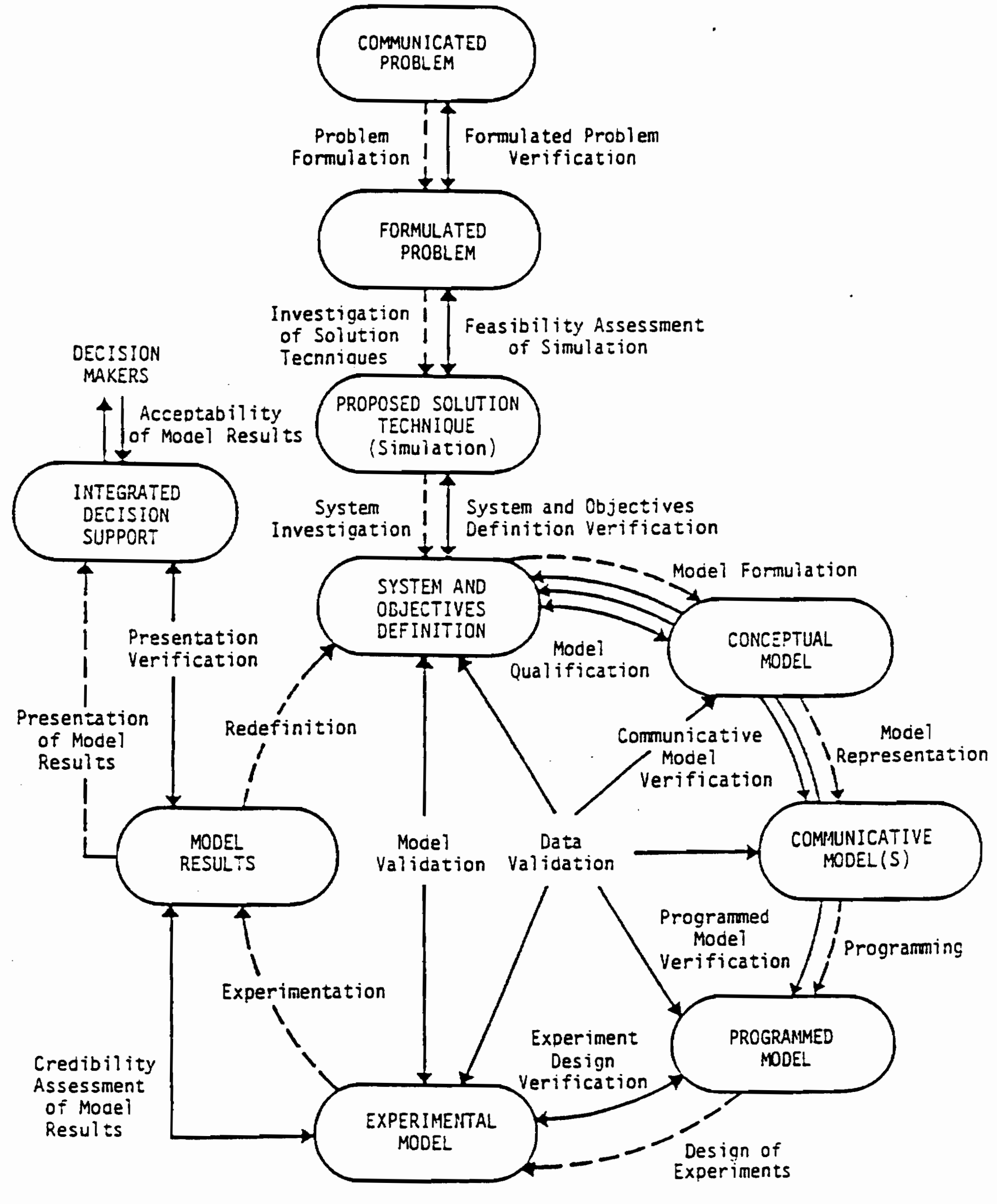

Figure 2. The Simulation Model Life-Cycle
[Nance 1981a; Balci 1986b]

Operational Approach. This approach has the advantage that the specification is itself an executable prototype. Analysis within the context of simulation model specification is facilitated by language classification according to the *basic descriptive unit* [Barger 1986]:

(1) Function - require that a system be defined as a series of functions which map inputs onto outputs. These languages enforce certain mathematical axioms thereby producing a *provably* correct specification [Martin 1985b].

(2) Object - yield a model representation composed of object descriptions containing the behavioral rules and characteristics for that object [Faught 1980].

(3) Process - in which a system is decomposed into processes, where a process may represent an object or an activity. Each process is assumed to operate in parallel with and asynchronous to the other processes [Zave 1984b].

A taxonomy of some (although certainly not all) popular and widely-used software specification languages is given in Table 3. The taxonomy originally appeared in [Barger 1986] and is updated here.

The simulation research community fully recognizes the need for model specification and model documentation; however, the precise nature of this need as well as the manner in which it can be met is not yet realized. How to represent model dynamics presents the main stumbling block in the design of any Simulation Model Specification and Documentation Language (SMSDL) since dynamic dependencies are inherently complex [Nance 1977, 1984]. Overstreet *et al.* [1986] provide a detailed review of the SMSDL's DELTA, GEST, ROSS, a SIMULA derived SMSDL proposed by Frankowski and Franta, and the Condition Specification (CS).

Table 2. The 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 1984; Zave 1984b]

Table 3. A Taxonomy of Specification Languages

BASIC DESCRIPTION UNIT	LANGUAGE	LIFE-CYCLE PHASE	UNDERLYING MODEL	REFERENCES
FUNCTION	ADTS	F,A,D,I	Mathematical	[Jadoul et al. 1989]
	Algebraic Spec	F	Mathematical	[Gehani 1982] [Liskov 1975]
	AXES	F,A,D	Mathematcal	[Hamilton 1976] [Hamilton 1983] [Martin 1985a] [Martin 1985b]
	Special	F	Mathematical	[Silverberg 1981] [Stoegerer 1984]
OBJECT	DDN	A	Message Passing	[Riddle 1978a] [Riddle 1978b] [Riddle 1979, 1980]
	MSG.84	F,A	Message passing	[Berzins 1985]
	PSL/PSA	F	ERA	[Stoegerer 1984] [Teichrow 1977] [Teichrow 1980] [Winters 1979]
	RDL	A,D	ERA	[Heacox 1979] [Stoegerer 1984]
	RML	F	ERA	[Borgida 1985] [O'Brien 1983]
	TAXIS	A,D	ERA	[Borgida 1985] [O'Brien 1983]
PROCESS	GIST	O	Stimulus Response ERA	[Balzer 1980, 1982] [Feather 1983] [Goldman 1980] [London 1982]
	HFSP	F,A	Stimulus Response	[Katayama 1989]
	PAISLey	O	Stimulus Response	[Yeh 1984] [Zave 1979, 1982] [Zave 1984a] [Zave 1984b]
	PDL	F,A	Stimulus Response	[Inoue et al. 1989]
	RSL	F,O	Stimulus Response	[Alford 1977, 1985] [Bell 1977] [Davis 1977] [Scheffer 1985] [Stoegerer 1984]

2.2.2 The Future

Even though formalism is a very powerful tool, the non-technical community, who represent the largest number of software sponsors, during the past decade has demonstrated an aversion to formal specifications. Clients who do not fully understand the specification of a proposed system cannot be expected to detect specification errors. This often results in the delivery of a flawed product.

In response to the lack of success of formal approaches, the software community shows signs of moving toward non-formal graphically-based system specifications. Many researchers share the opinion that graphically-oriented specification languages are the best approach to enhancing communication among a wide variety of specification audiences [Brackett 1988]. Much of today's software specification research moves in this direction [Yau and Jia 1988; Brown et al. 1985; Lodding 1982; Reiss 1986; Gillet and Kimura 1986]. Of note is the work of Harel [1988], which has resulted in STATEMATE, a product rapidly gaining popularity in industry. STATEMATE provides operational specifications combined with a graphical user interface. Execution of the model provides the user with a clear view of the system which may eliminate many of the communication problems associated with formal specification [Harel et al. 1988]. Other notable endeavors in the area of visual languages include: PECAN [Reiss 1985], HI_VISUAL [Ichikawa and Hirakawa 1987], and GARDEN [Reiss 1986] a graphical programming environment developed at Brown University.

2.3 User Interfaces

Since the user interface is *the* aspect of a software system that all users deal directly with, and since a poor user interface will result in an unusable and unsuccessful product (despite the quality and

functionality of the rest of the software), human factors considerations should be incorporated into the design of all systems at the initial stages of development [Badre et al. 1982; Yestingsmeir 1984]. A review of literature in this area indicates growing concern among the software engineering community for software designed to "adapt" to a user's information processing capabilities and limitations. The capacity and accessability of Short-Term Memory (STM), and many other human factors issues, are becoming major considerations in software development. Psychological studies reveal that the most-utilized memory processes occur in STM which holds information for around thirty seconds on the average. George Miller's [1956] survey indicates that information perceived by any sensory organ is limited to approximately seven units. For terminal interaction, these studies imply that the processing capacities of individuals are extremely small and in constant danger of overload [Shneiderman 1980]. However, psychology is not an exact science; one study on users of menu-driven software indicates that users prefer three separate menus rather than three menus on the screen at once, while the results of another study tend to indicate just the opposite, and yet another study indicates no user preference at all [Shneiderman 1987].

A study by Dumais [1982] examines whether an "easy-to-use" software system should be menu-selection based, keyword based, or natural-language based. The results indicate that natural-language based software is the easiest to learn and use but by far the most difficult to design. Unfortunately, an algorithm for optimal (or even satisfactory) user interface design is yet to be discovered [Shneiderman 1980].

Interactive system designers must obtain a viable compromise between conflicting design goals. Systems should be simple but powerful, easy to learn but appealing to experienced users, and facilitate error handling but allow freedom of expression. All of this should be accomplished in the shortest possible development time; costs should be kept low and future modification should be simple [Shneiderman 1980]. The difficulty in defining a universal set of goals for interactive systems arises due to the variety of situations in which systems are used. Figures 3 - 5 illustrate the design principles of three particular system designers. Although the era of the works is dated, the principles identified are still valid in the 1990's. Hansen's [1971] key is to "know the user." Gaines [1975]

agrees, stating "use the user's model." Each designer stresses consistency and flexibility, as well as explicit on-line assistance and diagnostics. Following such guidelines in the development of a human-computer interface is commonly considered a necessary but insufficient condition for constructing a good interface [Molich and Nielsen 1990]. Despite these guidelines, given any arbitrary set of users interacting with a system, some will find it unappealing. However, this fact should in no way prevent the application of design principles to the construction of user interfaces. In the absence of a *perfect* user interface, designers must apply solid design principles to achieve the best *usable* interface.

The consensus among system designers seems to be that a good software interface is sufficiently robust and flexible to accommodate a wide range of users. Procedures should be simple and easy to learn while at the same time permitting more experienced users to take "short-cuts" through the system. These factors seem to reflect the *usability* of the user interface.

User engineering principles.

First principle: Know the user.

Minimize memorization.
- Selection not entry.
- Names not numbers.
- Predictable behavior.
- Access to system information.

Optimize operations.
- Rapid execution of common operations.
- Display inertia.
- Muscle memory.
- Reorganize command parameters.

Engineer for errors.
- Good error messages.
- Engineer out the common errors.
- Reversible actions.
- Redundancy
- Data structure integrity.

Figure 3. User Engineering Principles for Interactive Systems [Hansen 1971]

Provide a program action for every possible type of user input.

Minimize the need for the user to learn about the computer system.

Provide a large number of explicit diagnostics, along with extensive online user assistance.

Provide program of short-cuts for knowledgeable users.

Allow the user to express the same message in more than one way.

Figure 4. The Design of Idiot-Proof Interactive Programs
[Wasserman 1973]

Introduce through experience.

Immediate feedback.

Use the user's model.

Consistency and uniformity.

Avoid acausality [*sic*].

Query-in-depth (tutorial aids).

Sequential - parallel tradeoff (allow choice of entry patterns).

Observability and controllability.

Figure 5. Design Guidelines for Interactive Systems
[Gaines 1975]

3 Model Generation

Model generation is the process of creating a model of some system which may be communicated to others and/or executed to produce behavior intended to represent that of the target system. Traditionally, model generation in discrete event simulation has been viewed as programming. A modeler first conceptualizes the system in his/her mind and then constructs a program of that system in a chosen language.

Unfortunately, the programming language representation of the model often fails to facilitate communication of the system to non-programmers such as managers. So, an English description is developed and/or diagrams constructed which allow communication between technical and nontechnical project personnel. This method contains severe flaws, the most obvious being that no one but the programmer knows if the English description *actually* reflects the program. A more subtle problem, but one which inflicts the greatest harm to simulations developed in this manner, is that a simulation programming language (or *any* language) imposes a world-view on the person using the language. A modeler, unaware of this, may choose a language inappropriate for the particular model simply because it is the simulationist's favorite (or the only one known). This strategy at best leads to inefficient and inelegant solutions, and at worse may prevent a solution altogether.

To alleviate the problems identified above, model *specifications* should be generated that are *independent* of any particular implementation world-view (programming language). These world-view independent specifications can be utilized as a communication medium and as an indicator for the selection of an appropriate programming language. The SMDE follows this philosophy with the Conical Methodology providing the structural underpinnings of the environment. Its greatest influence can be seen, by far, in the Model Generator.

3.1 The Conical Methodology

To facilitate the development of concepts in this section, as well as the remainder of this work, the following example of the classical machine repairman model which appeares in studies by Nance [1971, 1981a] and is from [Palm 1947; Cox and Smith 1961] is referenced:

> A single operator is assigned to service a group of n semiautomatic machines ($1 < n < \infty$) which fail intermittently and are repaired by the operator. Machine failure rates are assumed to follow a Poisson distribution with parameter λ. The repairman can service a failed machine in a time period that is exponentially distributed with parameter μ. On completing a machine service, the repairman determines if any machines are failed; if so, a failed machine is selected for repair (this selection algorithm has many variations - a first failed/first repaired discipline is assumed here). If all machines are operating at the conclusion of a repair, the repairman walks to a designated idle location to await the next failure. The travel times, both to machines and the idle location, are functionally determined by the identifiers of the two locations.

The following definitions are also needed. Fundamental is the concept of a **system** as defined in the Delta project report [Holbaek-Hanssen et al. 1977, p. 15]:

> A **system** is a part of the world which we choose to regard as a whole, separated from the rest of the world for some period of consideration, a whole which we choose to consider as containing a collection of components.

This system may be real or imagined and may have inputs and outputs which allow communication between the system and its environment [Overstreet 1982]. A **model** is an abstraction of a system intended to replicate some properties of that system. The level of detail and the type of abstraction depend on the properties the model is intended to replicate [Overstreet 1982]. According to Nance [1981b, p. 175] a model is comprised of **objects** and the relationships among objects. An object is anything that can be characterized by one or more **attributes** to which **values** are assigned.

3.1.1 Introduction to the Conical Methodology

Modeling is a creative process. To increase the chances of a *successful* result, every creative process requires foundational support to structure and form the creative activity. The scientific method provides the framework for experimental science; the principles of software engineering guide the process of software development. When the object of the design becomes complex, the framework is most essential. The human mind is fundamentally limited as to the size and scope of any particular problem that it can effectively handle. Systems are becomming more and more complex, and therefore modeling these systems is also increasing in complexity. A model of a single server queueing network can easily be solved without the aid of any real solution framework. However, the difference between this problem and the problem of modeling a global economy, say, is like the difference between the problem of adding two and two, and the problem of calculating the seventh root of a fifty digit number. In the first case, we can easily do it in our heads. In the second case, the complexity of the problem will defeat us unless we find a simple way of writing it down, which lets us break it into smaller problems [Alexander 1964].

For simulation modeling, one available framework is the Conical Methodology (CM) [Nance 1981a, 1987]. The CM is an object-oriented methodology that has been shown to be supportive of the principles of software engineering [Nance 1987]. The CM structures the modeling process

by defining two stages of model development. The CM provides a top-down model decomposition known as the *model definition phase,* and a bottom-up model synthesis refered to as the *model specification phase*. Although the stages are identified as separate processes, they are not meant to be executed either independently or without iteration. The CM encourages model development as a cycling through the two stages in a manner akin to successive refinement, and places no restrictions on a modeler other than the requirement that object definition must precede object specification [Nance 1981a].

3.1.2 Model Definition

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

The CM advocates strong typing to facilitate specification analysis. CM attributes are typed according to the taxonomy taken from Nance [1987] in Figure 6. To briefly summarize, *indicative* attributes describe an aspect of an object and may be classified as *permanent* or *transitional. Relational* attributes relate an object to one or more objects and may be classified as *hierarchical* or *coordinate*. A permanent indicative attribute can be assigned a value once, and only once, during model execution. Transitional indicative attributes come in two categories: *status* and *temporal.* Status transitional indicative attributes can be assigned a value from a finite set of possible values. Temporal transitional indicative attributes are assigned a value which is a function of time. Hi-

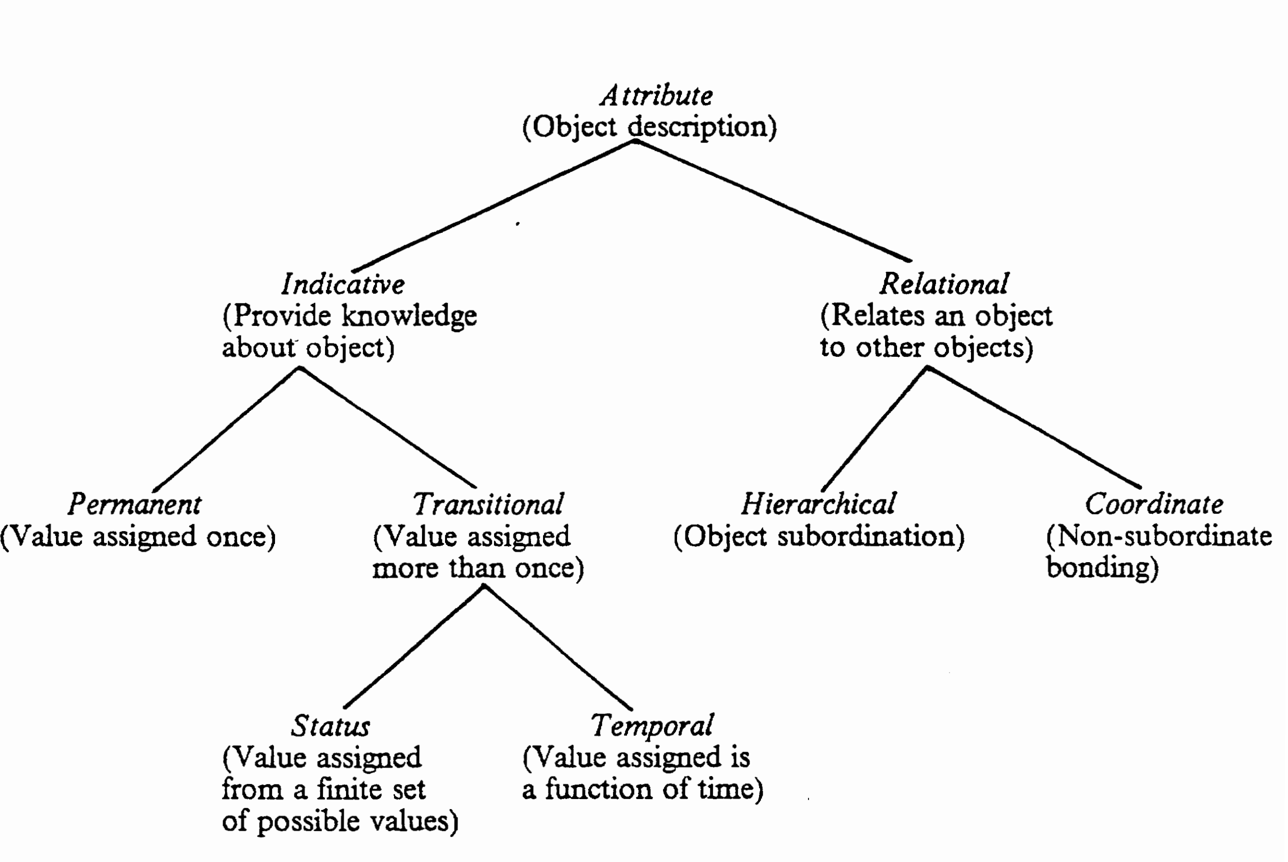

Figure 6. The Tree of Conical Methodology Types
[Nance 1987]

erarchical relational attributes establish the subordination of one object to another; whereas coordinate relational attributes establish a bond or commonality between two objects based on attributes of one object appearing in value expressions for another object. In addition to typing, the dimensionality and range of an attribute may also be provided during model definition.

At least one attribute is associated with every model: *system time* is an attribute of the top level model object. Its CM typing is always temporal transitional indicative; its dimensionality is user specified, and may range from simulation start time to simulation termination time. In the machine repairman model, the object ***repairman*** might have the attributes *status* - a status transitional indicative attribute which may take on the range of values (available, travel, busy), and *location* - a coordinate relational attribute which relates the position of the repairman to the machines. (Note that *location* might also be seen as a status transitional indicative attribute since its value reveals something about the repairman. The CM permits an attribute to have multiple types.)

3.1.3 Model Specification

The CM stipulates a bottom-up specification process. During the model specification phase, a modeler starts at the leaves of the model decomposition tree and describes what effect attributes of one object, by their value changes, have on attributes of other model objects thereby prescribing object behavior. In essence, during the model specification phase, a modeler identifies and describes model dynamics.

The CM as defined originally by Nance [1977] identifies the bottom-up specification process but provides little information about what form the model specification should take. Overstreet [1982] describes the Condition Specification (CS) which has been adopted by the CM as the target model specification. The CS satisfies the properties of a good simulation specification language [Nance 1977]:

(1) Exhibits independence of simulation programming languages,

(2) allows expression of static and dynamic model properties,

(3) facilitates model validation and verification, and

(4) produces documentation as a by-product.

In the machine repairman model, the (partial) specification of the attribute *status* of the object ***repairman*** may look like: "*status* of ***repairman*** equals *available* when end_of_repair is true." This partial specification details the state change of the object ***repairman*** by indicating how one of its status transitional indicative attributes, *status,* changes value from *busy* to *available.* Relationships in the model may also be specified such as: "*location* of ***repairman*** equals *location* of ***machine[i]*** when arrival_to_machine[i] is true." This specification establishes a relationship between the repairman and a failed machine through the use of coordinate relational attributes.

The specification given here is English-like and intended only to convey the essence of the specification process in the CM. The precise syntax and semantics of the Condition Specification are given in the following section.

3.2 The Condition Specification

A CS of a model consists of two components: a description of the communication interface for the model and a specification of the dynamics of the model [Overstreet 1982 p. 86]. The specification of model dynamics can be further divided into an object specification and a transition specification. The CS also identifies a report specification which provides details about statistical reporting of simulation results. The syntax for the Condition Specification is given in Table 4.

Table 4. 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> [, <argument list>])	Schedule an alarm
Cancel alarm	CANCEL ALARM(<alarm_name> [, <alarm id>])	Cancel scheduled alarm
When alarm	WHEN ALARM(<alarm_name expression> [(<parameter list>)])	Time sequencing condition
After alarm	AFTER ALARM(<alarm_name> & <boolean exp>, [(<parameter list>)])	Time sequencing condition
Create	CREATE(<object type> [, <object id>])	Generate new object
Destroy	DESTROY(<object type> [, <object id>])	Eliminate an object
Input	INPUT(<attribute_name>)	Assign value via input
Output	OUTPUT(<attribute_name>)	Produce output
Stop	STOP	End simulation

3.2.1 System Interface Specification

The system interface specification (originally refered to as the "interface specification"), identifies input and output attributes by name, data type and communication type (input or output). Overstreet [1982], assumes that the communication interface description can be derived from the internal dynamics of the model and can be system generated. According to [Overstreet and Nance 1985], any CS must have at least one output attribute.

3.2.2 Object Specification

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

3.2.3 Transition Specification

A transition specification consists of a set of ordered pairs called condition action pairs. Each pair includes a condition and an action to be associated with that condition. A condition is a boolean expression composed of model attributes and CS sequencing primitives, WHEN and AFTER. Actions come in five classes: a value change description, a time sequencing action, object generation (or destruction), environment communication (input or output), or a simulation run termination statement. The transition specification also includes a list of functions (if any) defined by a modeler to facilitate the description of model conditions.

Condition action pairs (CAPS) with equivalent conditions are brought together to form action clusters. Action clusters represent all actions which are to be taken in a model whenever the associated condition is true. Note that it is possible that many conditions in a model can be true simultaneously.

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

3.2.4 Report Specification

The syntax for report generation is undefined. Overstreet separates the report specification from the Condition Specification. This is not mandatory but often desirable [Overstreet 1982].

3.2.5 Example

A system interface specification, object specification, and transition specification for the machine repairman example are given in Figures 7-9 respectively. Similar examples have appeared in [Overstreet and Nance 1985; Nance and Overstreet 1988].

A short description of the transition specification given in Figure 9 is warranted. At initialization, all model parameters are input, the model object instances are created, and the initial failure for each machine is scheduled. Termination of the simulation occurs after a specified number of repairs have

been completed. When the repairman finishes repairing a machine (End repair action cluster), the next failure for that machine is scheduled, machine and repairman attributes are set to indicate that the machine is operational and the repairman available, and the number of repairs is incremented by one. These actions will cause either the condition "travel to idle" or "travel to facility" to become true, in turn dictating the direction the repairman will next travel. Interpretations of the other action clusters follow in a similar manner. Notice that incrementing number of repairs might also trigger the "termination" condition to evaluate to true. If that is the case, "termination" takes precedence over "travel to facility" or "travel to idle", and model execution is halted. Precedence for conditions is to be supplied by the modeler during the model analysis phase [Moose and Nance 1988].

Note the references to functions (e.q. neg_exp, traveltime, closest_failed_fac) in the transition specification. The CS does not provide specific guidelines for the specification of functions, but assumes that information such as the number of parameters and their types as well as the function type itself, will be provided in some form to facilitate analysis.

Input:

n	{ Number of machines }	positive integer
mean_uptime	{ λ }	positive real
mean_repairtime	{ μ }	positive real
max_repairs	{ # repairs for termination }	positive integer

Output:

{Percent uptime for each machine }	nonnegative real
{Average uptime for each machine }	nonnegative real

Figure 7. Machine Repairman System Interface Specification

Object	Attribute	Type
environment	system_time	positive real
	n	positive integer constant
	mean_uptime	positive real constant
	mean_repairtime	positive real constant
	max_repairs	positive integer constant
machine	id[1..n]	constant
	failure[1..n]	time-based signal
	failed[1..n]	boolean
repairman	status	(avail, travel, busy)
	location	(idle, id[i..n])
	end_repair	time-based signal
	arr_fac	time-based signal
	num_repairs	nonnegative integer
idle position	arr_idle	time-based signal

Figure 8. Machine Repairman Object Specification

```
{Initialization}
INITIALIZATION:
 VAR i:1..n;
 INPUT(n,max_repairs,mean_uptime,mean_repairtime);
 CREATE(repairman);
 FOR i := 1 TO n DO
   CREATE(machine);
   id[i] := i;
   failed[i] := false;
   SET ALARM(failure[i], neg_exp(mean_uptime));
 END FOR
 num_repairs := 0;
 location := idle;
 status := avail;
{Termination}
num_repairs ≥ max_repairs:
 STOP
{Failure}
WHEN ALARM(failure[i:1..n]):
 failed[i] := true;
{Begin repair}
WHEN ALARM(arr_fac[i:1..n]):
 SET ALARM(end_repair, neg_exp(mean_repairtime));
 status := busy;
 location := id[i];
{End repair}
WHEN ALARM(end_repair[i:1..n]):
 SET ALARM(failure[i], neg_exp(mean_uptime));
 failed[i] := false;
 status := avail;
 num_repairs := num_repairs + 1;
{Travel to idle}
(FOR ALL 1 ≤ i ≤ n, NOT failed[i]) &
status = avail & location ≠ idle:
 SET ALARM(arr_idle, traveltime(location, idle));
 status := travel;
{Arrive idle}
WHEN ALARM(arr_idle):
 status := avail;
 location := idle;
{Travel to facility}
status = avail & (FOR SOME 1 ≤ i ≤ n, failed[i]):
 VAR i:1..n;
 i := closest_failed_fac(failed,location);
 SET_ALARM(arr_fac, traveltime(location, fac[i]),i);
 status := travel;
```

Figure 9. Machine Repairman Transition Specification

3.3 Model Generator Prototypes

Because the Model Generator is identified as a crucial tool of the SMDE, substantial research has focused on that tool. Prior research efforts have led to three prototype Model Generators: A prototype designed by Balci and implemented by Bishop [1989], provides a vehicle for exploring the development of a new conceptual framework. Two earlier prototypes, one developed by Hansen [1984] and the other by Barger [1986], implement the Conical Methodology as a CF. The latter two have notable deficiencies. Hosted in UNIX[1] on a VAX 11/785[2], the Hansen and Barger prototypes are limited by their development within a "dumb terminal" interface and the absence of effective database support.

Nance *et al.* [1984] evaluate UNIX as a host for prototype SMDE's, noting major and minor deficiencies. Balci [1987] defines a new prototype SMDE based on a SUN-3/160[3] color computer workstation which overcomes many of these deficiencies. The UNIX 4.2BSD operating system supports several utilities in establishing the environment: multiwindow display manager (SunWindows), device independent graphics library (SunCore), computer graphics interface (SunCGI), visual integrated environment (SunView), Sun programming environment (SunPro), and INGRES relational database management system (SunINGRES).

The Balci-Bishop prototype runs on the SUN-based SMDE. The Model Generator described in this research is also targeted for the SUN and to utilize the improved capabilities permitted by the features identified above.

---

[1] UNIX is a trademark of AT&T Bell Laboratories.
[2] VAX 11/785 is a trademark of Digital Equipment Corporation.
[3] SUN-3/160 is a trademark of Sun Microsystems, Inc.

3.3.1 Box-Hansen

The Box-Hansen model generator represents the first prototype model generator of the SMDE. Box-Hansen implements the top-down model definition phase of the CM. The generator provides an interactive dialogue through which a modeler may define the objects that comprise a model, and attach attributes to those objects. The Box-Hansen prototype also allows the definition of sets within a model. However, the prototype offers no assistance by way of structure to the model specification phase.

3.3.2 Barger

The Barger prototype model generator is designed as an extension of Box-Hansen. Designed to use Box-Hansen as a front-end model definition driver, the Barger prototype provides a leveled dialogue approach to derive the specification of indicative attributes. The CS provides the target specification language for Barger's prototype. Barger's experimentation with the prototype indicates that the specification of status transitional indicative attributes may provide the key to the derivation of a complete specification since a modeler is able to provide the foundation of a model specification by describing the value changes of these attributes. Though designed to use Box-Hansen as a front-end, the two prototypes were never connected. Barger's prototype also suffers from an incomplete implementation of the CS.

3.3.3 Balci-Bishop

Bishop [1989] describes the General Purpose Visual Simulation System (GPVSS). GPVSS is a microcosm of the SMDE; it provides model definition, model specification, and model translation

into executable code, as well as animation of the simulation. The CF driving GPVSS is based on a graphical, object-oriented, and activity-based approach. GPVSS identifies a model as being composed of submodels which contain static objects and through which dynamic objects travel in a simulation. Submodels, and objects are defined graphically using a graphical editor. The paths that dynamic objects take through the simulation are also defined graphically.

Within GPVSS, the specification of model behavior is extremely low-level: the modeler must utilize C-based macros and in most cases provide C code. GPVSS falls short of the requirements for the model generator in the SMDE in that it is unclear how the analysis of the C specification may be facilitated, and the documentation produced by GPVSS is limited. It does, however, offer many desirable features, among them, graphical model definition. Ongoing research seeks to enhance GPVSS and provide greater structure to the specification by the definition of a specification language for GPVSS, as well as a refinement of the conceptual framework on which GPVSS is built.

3.4 Motivation for a New CM-Based Prototype

The work of [Nance and Overstreet 1984, 1988; Moose and Nance 1988] has shown the Condition Specification to offer a high degree of analyzability as well as world-view independence. For this reason, a prototype which facilitates the derivation of a CS model specification from a CM model definition is desirable. Previous prototype efforts have made significant gains in the realization of this goal, however no existing prototype fully captures all of the required features for the Model Generator in the SMDE. The limitations of previous prototypes, mentioned earlier, are outlined here.

3.4.1 "Dumb-Terminal" Interface Limitations

Both the Box-Hansen and Barger prototypes utilize a menu-driven, text entry "dumb-terminal" interface. New technology provides systems (such as SUN workstations) for which window-based, highly iconic software can be developed. Interfaces designed under this umbrella have a higher potential to satisfy the criteria for good user interfaces outlined in Section 2.3 than do interfaces which rely strictly on text entry.

3.4.2 Database Limitations

The move of the SMDE to the SUN-3/160 provides a significant step up from file management under UNIX System V to SunINGRES relational database technology. The relational approach represents the dominant trend in database research. According to Date [1976 p.20]:

> Almost all current database research is based on relational ideas. In fact, there can be little doubt that "the relational model" is the single most important development in the entire history of the database field.

The implications of relational database technology for model development are many. The relational approach allows the exploitation of submodel reuse, as well as readily facilitating submodel expansion and deletion from any level of the model decomposition tree. A prototype is desired which takes advantage of this technology.

3.4.3 Incomplete Specification

Barger's prototype provides only for the specification of indicative attributes. While this is a significant contribution, a prototype is desired which fully provides for the derivation of a condition

specification in order to adequately assess the capabilities of the CM/CS as a conceptual framework for the SMDE Model Generator.

3.5 Summary

Model generation is the process of creating a model of some system which may be communicated to others and/or executed to produce behavior intended to represent that of the target system. The role of the methodology is key to the model generation process. The Concial Methodology provides an object-oriented framework for model generation. The CM defines a two-phase model construction process. During the *model definition phase,* the modeler identifies the objects in the model and defines thier attributes -- effectively describing the *static* aspects of the model. During the *model specification phase,* the modeler identifies object interactions via value changes in object attributes -- effectively describing the *dynamic* aspects of the model. The Condition Specification provides a framework for model specification within the CM. Central to the CS is the condition action pair. Several prototype Model Generators have been constructed within the SMDE. The prototypes based on the CM suffer from many weaknesses, such as the lack of effective user interface and database support, as well as an incomplete treatment of the CM/CS.

4 Design of a New Model Generator Prototype

This chapter details the design of the current Model Generator prototype (hereafter referred to simply as the Model Generator). The Model Generator builds on the concepts developed in the work of Hansen [1984] and Barger [1986], fully integrating the capabilities of the two prototypes under the new target host system (the SUN workstation). This chapter focuses on the technical and conceptual *extensions* of the earlier prototypes. The reader is assumed to be familiar with the earlier research; therefore a discussion of specific details of integrating the Hansen and Barger prototypes and their respective features is omitted.

Section 4.1 describes the implementation of the Condition Specification within the Model Generator. Section 4.2 outlines the Model Generator's treatment of sets. The effects of definition changes on specification completeness are discussed in Section 4.3. The database is described in Section 4.4. The details of the Model Generator interface are given in Section 4.5, and Section 4.6 presents a summary of the design.

4.1 Specifying a Model under the CM/CS Framework

The target specification for the Model Generator is the CS. The primary output of the Model Generator in the SMDE, a CS model specification in the form of condition action pairs, is intended to be passed to the Model Analyzer. Although the CS is the form for the model specification, explicit knowledge of the syntax and semantics of the CS is embedded within the Model Generator, and *is not* required of the modeler. A modeler interacting with the Model Generator need only be familiar with the methodology (the CM) and the *concepts* utilized within the CS.

Barger [1986] identifies two possible approaches for obtaining condition action pairs from a modeler. The first approach is to allow a modeler to describe each condition and then the actions that are associated with it. This approach provides considerable freedom to a modeler, but does not result in a natural progression from the model definition phase to model specification. The second approach, and the one adopted by Barger for specifying status attributes, is to allow a modeler to select an attribute and a value change for that attribute (implicitly form the action <attribute> := <new_value>), *then* request that the modeler describe the condition(s) causing this value change. This latter approach seems to best utilize the information provided in the model definition phase, and affords a smooth transition into model specification. The Model Generator uses this approach and extends it to all components of the CS. In this section, we examine the aspects of model specification addressed by the Model Generator.

The Model Generator treats an attribute as local to the object for which it is defined. Accordingly, no constraint guarantees the global uniqueness of attribute names. This can lead to ambiguity under the CS syntax given in Table 4, in which attribute names are not associated with "owning" objects. To disambiguate, the Model Generator adopts a so-called "dot" notation when referring to attributes by dot-prepending the owning object as: <object_name>.<attribute_name>. Also,

if multiple instances of an object are created they can be differentiated as <object_name>(id).<attribute_name>.

An important note is that the CS syntax adopted by the Model Generator is based solely on the requirements of the Model Generator. Ideally, both the Model Generator and the Model Analyzer requirements should jointly dictate the CS syntax. Future prototypes should reflect this evolution.

4.1.1 Indicative Attributes

Barger [1986] proposes that status transitional indicative (sti) attributes provide the key to the derivation of a complete model specification. Since the values of sti attributes can be enumerated (from a finite set of possible values), a modeler may generate most, and possibly all, model conditions when describing those conditions that cause sti attributes to change values. Barger offers the technique in Figure 10 for the specification of status attributes. For a detailed analysis of this technique, refer to [Barger 1986]. Note that Barger refers to the "state" of an attribute. Strictly speaking, only objects have states, i.e. the state of an object is determined by the enumeration of the values of its attributes at any given time. In this work, we refer to attributes as having "value" changes rather than "state" changes.

4.1.1.1 Status Transitional

The algorithm for specification of status transitional indicative attributes is given in Figure 11. In this process, an sti attribute is selected, its values enumerated (if possible), and the condition(s) causing each value change identified. One *special* case of status transitonal indicative attribute exists: the case of the counting variable. In the Conical Methodology work [Nance 1981a, 1984, 1987], a modeler must exhaustively enumerate the values a status transitional indicative attribute may assume. This enumeration facilitates the specification of the attribute since the modeler is

Step 1: Given a defined model, select a status attribute.

Step 2: List all values which may be assigned to the status attribute. Call each value a state.

Step 3: For each state in step 2, list the successor state. Call the ordered pair (state, successor state) a state change.

Step 4: Select a state change. Form the action
attribute name := successor state

Step 5: Specify a condition that causes the state change.

Step 6: If the condition is time-based, go to step A. Otherwise go to step 7.

Step 7: Repeat steps 5 and 6 until all conditions causing that state change have been specified.

Step 8: Repeat steps 4-7 until all state changes for that status attribute have been done.

Step 9: Repeat steps 1-9 until all status attributes have been selected.

Step A: Specify a condition that causes the alarm to be set.

Step B: If the condition is time-based, but involves a different alarm, repeat steps A-D with the new alarm.

Step C: If the condition is state-based, return to step A and repeat with another condition.

Step D: Repeat steps A-D until all conditions that cause the alarm to be set have been specified.

Step E: Return to step 7.

Figure 10. Barger's Technique for Status Attribute Specification [Barger 1986, p. 52]

Step 1: Select a defined status transitional indicative attribute.

Step 2: Enumerate (where possible) all values that may be assigned to the attribute.

Step 3: For each enumerated value in step 2, list the successor value. Call the ordered pair (value, successor value) a value change. (If step 2 yields a range, form the value change: (n, n + 1))

Step 4: Select a value change, form the action:
object_name.attribute_name := successor value

Step 5: Specify a condition that causes the value change. If the condition is previously defined, form the CAP and goto step 9, else continue.

Step 6: Name a new model condition.

Step 7: Describe the new condition.

Step 8: If the condition is state-based, enter the boolean expression for the condition and form the CAP, else goto Specify Alarm and form the CAP on return.

Step 9: Repeat step 5 until all conditions for the value change have been specified.

Step 10: Repeat steps 4-9 until all value changes have been specified.

Step 11: Repeat steps 1-10 until all sti attributes have been specified.

Figure 11. Algorithm for Status Transitional Indicative Attribute Specification

easily led to describe the conditions precipitating each value change. However, for a counting variable (one that ranges over the nonnegative integers, say) it is not desirable, or even possible in most cases, to identify a condition that causes the attribute to change from 0 to 1, and then from 1 to 2, 2 to 3, and so on *ad infinitum.* To accommodate this anomaly, the Model Generator requests, but *does not force,* the enumeration of status transitional indicative attribute values. For an sti attribute that is defined over a range rather than enumerated, the value change from n to $n+1$ is recommended as the value change to be specified. As part of model analysis, the Model Generator checks the list of values defined for an sti attribute against the values within the value change being specified, identifying any inconsistencies to the modeler, but the Model Generator *does* recognize and the counting variable as a special case.

The algorithm for status transitional indicative attribute specification refers to the algorithm Specify Alarm which appears in Figure 14. This algorithm derives the appropriate CS primitive for a time-based or mixed (time-based and state-based) condition. Specify Alarm is called from several locations; therefore it appears as its own algorithm rather than embedded within the sti attribute algorithm (as is the case in Barger [1986]).

4.1.1.2 Temporal Transitional

The algorithm for specification of temporal transitional indicative (tti) attributes is given in Figure 12. In this process, a tti attribute is selected, a range is provided (if known *a priori*), and the right-hand side of the assignment action is determined. A tti attribute may be assigned a value as a result of an attribute-valued expression, the return of a function call, or the value may be input. In any case the assignment must yield a value that is a function of time.

Step 1: Select a defined temporal transitional indicative attribute.

Step 2: Specify a range of possible values (if known).

Step 3: Determine the value to be assigned to the attribute.
Value may be an attibute-valued expression or function call on RHS of assignment, form the action: object_name.attribute_name := RHS
Value may be input, form the action: INPUT(object_name.attribute_name)

Step 4: Specify a condition that causes the attribute to be assigned a value.
If the condition is previously defined, form the CAP and goto step 8, else continue.

Step 5: Name a new model condition.

Step 6: Describe the new condition.

Step 7: If the condition is state-based, enter the boolean expression for the condition and form the CAP, else goto Specify Alarm and form the CAP on return.

Step 8: Repeat step 4 until all conditions have been specified.

Step 9: Repeat steps 1-8 until all tti attributes have been specified.

Figure 12. Algorithm for Temporal Transitional Indicative Attribute Specification

4.1.1.3 Permanent

The algorithm for specification of permanent indicative (pi) attributes is given in Figure 13. In this process, a pi attribute is selected, a range is provided (if known *a priori*), and the value to be assigned the attribute is determined. A pi attribute may be assigned a value as a result of an attribute-valued expression, a function call, or it may receive a value as a result of model input. A permanent indicative attribute may receive a value *only once* during model execution. For this reason, the Model Generator forces the constraint that the condition causing value assignment is executed only once.

While this restriction on permanent attribute assignment can be made for non-interactive execution, such a provision is not possible for interactive model execution when a modeler can manipulate pi attributes in *any* modeler-defined condition. In this case the Model Generator must assure that the actions in the associated action cluster affect *only* permanent indicative attributes. However, this type of static analysis cannot guarantee that the condition can execute only once. Accordingly, the Model Generator allows only the conditions Initialization and Termination to manipulate permanent indicative attributes. The class of discrete event simulation models is believed to be unaffected by this constraint.

4.1.2 Relational Attributes

The CM provides the means to represent relations among objects via attributes. This approach differs from that of other modeling methodologies such as the ER approach [Chen 1976] where relationships exist as concepts separate from attributes. Capturing relationships via attributes, however, may benefit model analysis [Moose and Nance 1988].

Step 1: Select a defined permanent indicative attribute.

Step 2: Specify a range of possible values (if known).

Step 3: Determine the value to be assigned to the attribute.
Value may be an attibute-valued expression or function call on RHS of assignment, form the action: object_name.attribute_name := RHS
Value may be input, form the action: INPUT(object_name.attribute_name)

Step 4: Specify a condition that causes the attribute to be assigned a value.
Only one (1) condition may do so and that condition must be Initialization or Termination.

Step 5: Form the CAP.

Step 6: Repeat steps 1-5 until all pi attributes have been specified.

Figure 13. Algorithm for Permanent Indicative Attribute Specification

Step 1: Get alarm to be associated with a condition.
If alarm is new, attach as an attribute to the top level model object,
CM type is temporal transitional indicative, CS type is time-based signal.

Step 2: If condition is time-based, form the condition expression:
WHEN ALARM(alarm_name [,parameter])

Step 3: If the condition is mixed, enter the boolean expression, form
the condition expression:
AFTER ALARM(alarm_name AND bool_exp [,parameter])

Step 4: Return.

Figure 14. Algorithm for Alarm Specification

The CM allows an attribute to have multiple types, i.e. a single attribute may be both indicative (in that it describes an aspect of an object) and relational (in that it relates one object to another). The research for this thesis raises the conjecture that nothing is gained from a model specification perspective by typing an attribute as *strictly* relational. The Model Generator's treatment of sets provides many of the relations required for model specification and may subsume the need for explicit relational typing. Further, the conjecture is offered that relations among (non-set) objects can liklely be deduced from indicative attributes. However, a major tenet of the SMDE is to provide methodological *guidance* without coercion or overbearing constraints. Therefore, the Model Generator allows a modeler to use relational attributes in whatever manner seems natural to the modeler, even though the use of relational attributes to define model relationships may, in some sense, provide redundant information.

The algorithm for relational attribute specification appears in Figure 15. This algorithm should be executed only for purely relational attributes. If an attribute has indicative typing, the appropriate indicative specification algorithm should be used to specify the attribute. A discussion of the classes of relational attributes follows.

4.1.2.1 Coordinate

According to Nance [1981a p. 27] a coordinate relational attribute:

> establishes a bond or commonality between two objects based on: (1) the value(s) of one or more attributes, or (2) the appearance of one (or more) attribute(s) of one object in the expression for value assignment to one (or more) attributes of the other object.

In the example from Chapter 3, a coordinate relational attribute establishes the relation between the repairman location and a failed machine under repair, i.e. the repair of a machine requires that the attribute *location* of the object ***repairman*** take on the value of the identifier of the failed machine.

Step 1: Select a defined (purely) relational attribute.

Step 2: Specify a range of possible values (if known).

Step 3: Determine RHS of assignment action. RHS may be an attribute valued expression or a function call. Form the action:
object_name.attribute_name := RHS

Step 4: Specify a condition that causes the attribute to be assigned a value. If the condition is previously defined, form the CAP and goto step 8, else continue.

Step 5: Name a new model condition.

Step 6: Describe the new condition.

Step 7: If the condition is state-based, enter the boolean expression for the condition and form the CAP, else goto Specify Alarm and form the CAP on return.

Step 8: Repeat step 4 until all conditions have been specified.

Step 9: Repeat steps 1-8 until all relational attributes have been specified.

Figure 15. Algorithm for Relational Attribute Specification

As stated above, the designation of *location* as a relational attribute, although perhaps useful, may not be necessary. Certainly, *location* can be viewed as providing information about the object ***repairman.*** Since the value of *location* both varies during model execution and indicates something about the state of the ***repairman,*** its type could well be status transitional indicative. Clearly the relationship between ***repairman*** and ***machine*** is evident in the expression: *repairman.location := machine.identifier*. The presence of an indicative attribute of one object on the right-hand side of an assignment expression for an indicative attribute of another object implies a relationship among the objects.

Further research into the types of analysis that can be performed on a model specification in a CS form may answer the question of whether relationships among objects should be explicitly defined via relational attributes or implicitly captured with indicative attributes. In the absence of a definitive answer to this question, the Model Generator permits either approach.

4.1.2.2 Hierarchical

According to Nance [1981a p. 27] an hierarchical relational attribute:

> establishes a subordination of one object to another, implying that all characteristics of the subordinate object are descriptive of the superior object.

Nance further indicates that the need for hierarchical relational attributes arises when using sets in the CM. These hierarchical relationships allow the differentiation of the set object and set member objects, as well as providing indexing into the set. Experiments with the Model Generator indicate that a modeler simply needs the ability to manipulate the objects within a set, via operations like Insert, Delete, Member, etc., and does not specifically need to know how set access is being achieved through various heirarchical relationships. Hence, the facility for a modeler to define an attribute as hierarchical relational may not be required.

Note that a modeler may not choose to use sets in model specification. Accordingly, insufficient evidence exists to conclude that hierarchical relational typing is unnecessary. Consequently, the Model Generator enables a modeler to type an attribute as hierarchical relational. Further details of heirarchical relationships are provided in the discussion of set specifcation in Section 4.2.

4.1.3 Time-Based Signals

Time sequencing in a CS is provided via *alarms*, or time-based signals, which are set by the primitive SET ALARM to "go off" at some value of system time. We have already seen (in Figure 14) how the Model Generator deduces the WHEN ALARM and AFTER ALARM conditions when matching a condition to some model action. Of course, every alarm must have a SET ALARM action in order to allow the execution of its corresponding WHEN ALARM or AFTER ALARM. The algorithm for time-based signal specification is given in Figure 16. An important distinction to note is that the algorithm for alarm specification derives a *condition* (a WHEN ALARM or AFTER ALARM). This algorithm derives an *action* (a SET ALARM) and the condition which precipitates it. Of course, the condition for a SET ALARM action may consist of a WHEN ALARM or AFTER ALARM for the same, or a different, time-based signal.

The CS primitive CANCEL ALARM is not provided by the Model Generator since this function is believed to be nonessential for representing the class of discrete event simulation models [Zeigler 1976]. We speculate that CANCEL is required only for "man-in-the-loop" simulations.

The Barger prototype forces the modeler to attach a time-based signal to an object as an attribute of the object. Barger [1986 p.116] raises the question, "should an alarm be attached to multiple objects...the initial answer seems to be 'yes'." Barger's theory is that an alarm should be an attribute of any object it "affects," i.e. any object that appears in the action cluster cooresponding to a WHEN ALARM or AFTER ALARM condition for a particular alarm. This many-to-many re-

lationship, however, may have negative effects on model analysis, and definitely clutters the alarm specification process, often causing a modeler to digress far from the original specification goal. The solution offered by the Model Generator is that all alarms are *automatically* attached as attributes of the highest level object (the model). Since all alarms are attributes of the same object, alarm names are unique. By utilizing the object-oriented concept of inheritence, all objects belong to the class that is defined by the model. Therefore, all objects have access to the attributes of the model object. This solution reduces confusion in the model specification process and provides all model objects with access to *system_time,* as well as the model alarms. Note, that in an object-oriented context, alarms are the message-passing mechanism of a condition specification. In some sense, this mechanism must be global (at least among all communicating objects). Further note that this representation serves only the specification and may likely bear little resemblance to an implementation which may encapsulate the alarms to within multiple objects.

4.1.4 Model Input and Output

The algorithm for specification of model input and output (I/O) is given in Figure 17. This facility provides flexibility to a modeler in that it allows a modeler to specify model I/O directly without using the (more involved and "leading") attribute specification dialogues described earlier.

4.1.5 Object Creation and Destruction

The algorithm for object creation is given in Figure 18. This rather straightforward algorithm allows a modeler to select an object and the condition that causes its creation or destruction. Note that the CM/CS stipulates that any object defined must have *at least* one attribute.

Step 1: Select a defined time-based signal.

Step 2: Determine the time alarm is set to "go off."
Value may be an attribute-valued expression, or a function call.
Form the action:
SET ALARM(alarm_name,value)

Step 3: Specify a condition that causes the alarm to be set.
If the condition is previously defined, form the CAP and goto step 7,
else continue.

Step 4: Name a new model condition.

Step 5: Describe the new condition.

Step 6: If the condition is state-based, enter the boolean expression for
the condition and form the CAP, else goto Specify Alarm and form
the CAP on return.

Step 7: Repeat step 3 until all conditions have been specified.

Step 8: Repeat steps 1-7 until all time-based signals have been specified.

Figure 16. Algorithm for Time-Based Signal Specification

Step 1: Select a defined object.

Step 2: Select an attribute of the object, and I/O operation.
Form the action: INPUT(object_name.attribute_name) or
OUTPUT(object_name.attribute_name)

Step 3: Specify a condition that causes the attribute to be input or output.
If the condition is previously defined, form the CAP and goto step 7,
else continue.

Step 4: Name a new model condition.

Step 5: Describe the new condition.

Step 6: If the condition is state-based, enter the boolean expression for
the condition and form the CAP, else goto Specify Alarm and form
the CAP on return.

Step 7: Repeat step 3 until all conditions have been specified.

Step 8: Repeat steps 1-7 until all model input/output has been specified.

Figure 17. Algorithm for Model Input and Output Specification

Step 1: Select a defined object. [Select an identifier.]

Step 2: Select an action.
Form: CREATE(object [,identifier]) or DESTROY(object, [,identifier])

Step 3: Specify a condition that causes the object to be created or destroyed.
If the condition is previously defined, form the CAP and goto step 7,
else continue.

Step 4: Name a new model condition.

Step 5: Describe the new condition.

Step 6: If the condition is state-based, enter the boolean expression for
the condition and form the CAP, else goto Specify Alarm and form
the CAP on return.

Step 7: Repeat step 3 until all conditions have been specified.

Step 8: Repeat steps 1-7 until creation of all objects has been specified.

Figure 18. Algorithm for Object Creation and Destruction Specification

4.1.6 Model Initialization and Termination

The algorithm for the specification of model initialization is given in Figure 19. No expression is associated with the initialization condition; initialization is true only at the beginning of a model instantiation. When specifying initialization a modeler must provide an initial value for system time (and that value shall be a constant). For any p-set in the model, a modeler must provide at least one expression for a permanent indicative attribute for each object in the p-set. When a p-set is defined, the Model Generator automatically generates a For-loop to be executed during initialization that creates each of the *n* instances of the p-set object. The loop takes the form:

```
FOR i := 1 to <number in p-set> DO
     CREATE(p-set_object);
END FOR;
```

A modeler needs to provide an assignment action of the form: *p-set_object.<permanent indicative attribute name> := i;* in order to distinguish between the objects in the p-set. When specifying initialization, the Model Generator requests that the modeler indicate which permanent indicative attribute identifies p-set objects. The Model Generator constructs the assignment action and places it in the For-loop immediately following the CREATE statement. A modeler may also provide initial values for any model attributes via assignment actions, specify object creation, model input, and set alarm actions during initialization specification.

The algorithm for termination specification is given in Figure 20. When specifying termination a modeler must provide an expression for the condition. The expression for termination must be boolean and may not involve an alarm. The action STOP is automatically provided by the Model Generator. A modeler may also specify model output and attribute assignment at termination.

Must: Provide an initial (constant) value for system time.
Form action: model_name.system_time := value

Must: For any p-set, provide an assignment to a permanent indicative attribute. This should be done in the context of the p-set object creation loop provided by the Model Generator.

May: Specify initial values for model attributes.

May: Specify CREATION action for any model object.

May: Specify SET ALARM action for any model alarms.

Figure 19. Algorithm for Initialization Specification

Must: Provide a boolean expression for termination.

May: Specify assignment action to any model attribute.

May: Specify OUTPUT action for any model attribute.

Figure 20. Algorithm for Termination Specification

4.1.7 User-Defined Functions

The ability to define functions when specifying a model is vital. The overriding question is how much function information should be provided during model generation, and what form should the information take? One possibility is to allow a modeler to code the function, but in what language? Since the underlying executable target language for simulations developed in the SMDE is as yet undefined -- and the SMDE is likely to provide a variety of simulation programming languages that can be generated from a single model specification -- no clear choice exists for a function specification language. Ideally, of course, we would like to specify the function in the same specification language that is used to specify the model, but since the CS does not provide a specific mechanism for function specification, the Model Generator provides only for the definition of functions in terms of function name, parameters, and types. The algorithm for function specification is given in Figure 21.

4.2 Representing Sets in Model Specification

The CM identifies two types of sets [Nance 1981a, p. 28]:

> A primitive set (p-set) is an object representing a collection of objects all of which have identically the same attributes.
>
> A defined set (d-set) is an object representing a collection of objects, not necessarily having the same attributes, defined by an expression evaluation during model execution.

The property of set membership in a p-set is static, therefore p-sets can be completely described prior to model execution. When defining a p-set with the Model Generator, a modeler must provide the number in the p-set as well as the attributes of the p-set which comprise the attribute set for each of the members of the p-set. As mentioned in Section 4.1.6, the modeler must provide *at least* one permanent indicative attribute for a p-set. This pi attribute, typically viewed as an iden-

Step 1: Name the function.

Step 2: Provide function return type (CS typing).

Step 3: Describe the function.

Step 4: Provide a function parameter.

Step 5: Type the parameter (CS type).

Step 6: Repeat steps 4-5 until all parameters have been defined.

Step 7: Repeat steps 1-6 until all functions have been defined.

Figure 21. Algorithm for Function Specification

tifier, is required in any p-set to distinguish uniquely among set members. D-set membership for model objects is contingent on the value(s) of some nonempty subset of the object's attributes at any time during model execution.

Set manipulation in the Model Generator is provided via the typical mathematical set operations, INSERT, DELETE, MEMBER, and FIND, with the usual semantics. The syntax and function of these operations is given in Table 5. A modeler must specify any set activity in the set specification dialogue. The algorithm for set specification is given in Figure 22. Since p-set membership is static, only d-sets may be manipulated with the INSERT and DELETE operations. Of course, a d-set could be a subset of a p-set and in that sense p-set objects *can* be manipulated, but the cardinality of p-sets is fixed throughout model execution. During set specification, a modeler must provide the condition(s) for set membership (if any exist) of each model object for any and all d-sets defined for the model. Note that the set operations MEMBER and FIND can be referenced in any expression for a model condition or action.

4.3 Specification Completeness

One of the questions raised by the research into the development of the current Model Generator is in the area of specification completeness. What types of analysis should be provided in the Model Generator, and to what extent? This philisophical question is related to the principle of "separation of concerns" identified by [Dijkstra 1972]. The Model Analyzer provides, and *should* provide the bulk of model analysis within the SMDE framework. A thorough questioning of the role of analysis within the Model Generator is therefore mandated.

A simple criteria for specification completeness analysis within the Model Generator is the availability of all specification information. If all specification information is present in the database,

Table 5. Syntax and Function of Set Operations

<table>
<tr><th>NAME</th><th>SYNTAX</th><th>FUNCTION</th></tr>
<tr><td>Insert</td><td>INSERT(<object_name> [(id)],
<set_name>)</td><td>Object insertion</td></tr>
<tr><td>Delete</td><td>DELETE(<object_name> [(id)],
<set_name>)</td><td>Object deletion</td></tr>
<tr><td>Member</td><td><boolvar> := MEMBER(<object_name>
[(id)], <set_name>)</td><td>Set membership</td></tr>
<tr><td>Find</td><td><setvar> := FIND(FIRST | LAST |
ALL | UNIQUE,
<set_name>, <expression>)</td><td>Find object or set
based on expression</td></tr>
</table>

Step 1: Select a model object. [Select object identifier.]

Step 2: Select a d-set.

Step 3: Form the action: INSERT(object_name[(id)], set_name)

Step 4: Specify a condition that causes the object to be inserted into the d-set. If the condition is previously defined, form the CAP and goto step 13, else continue.

Step 5: Name a new model condition.

Step 6: Describe the new condition.

Step 7: If the condition is state-based, enter the boolean expression for the condition and form the CAP, else goto Specify Alarm and form the CAP on return.

Step 8: Form the action: DELETE(object_name[(id)], set_name)

Step 9: Specify a condition that causes the object to be deleted from the d-set. If the condition is previously defined, form the CAP and goto step 14, else continue.

Step 10: Name a new model condition.

Step 11: Describe the new condition.

Step 12: If the condition is state-based, enter the boolean expression for the condition and form the CAP, else goto Specify Alarm and form the CAP on return.

Step 13: Repeat step 4 until all conditions for object insertion have been specified.

Step 14: Repeat step 9 until all conditions for object deletion have been specified.

Step 15: Repeat steps 2-14 until all d-sets have been specified for the current object.

Step 16: Repeat steps 1-15 until all objects have been addressed.

Figure 22. Algorithm for Set Specification

the Model Generator assumes a complete specification. Problems arise in this naive approach, however, when a definition change occurs in a previously *complete* model specification.

Once a specification is flagged as complete, what effect does a change in model definition have on the completeness of the specification? The immediate answer is unclear. Initially, the hope was that definition changes could be isolated to affect the specification of model objects falling below the modified object in the model hierarchy, and only those objects. However, experimentation indicates that a definition change of any object could potentially affect the specification of any other model object at any level in the model hierarchy. Therefore, when a definition change occurs, for example an attribute is deleted from an object, the entire model specification must be scanned and *all* conditions and actions affected by the change identified to the modeler, who must respecify the model to accommodate the change.

Beyond the test that all specification information is present in the model database, the Model Generator assesses specification completeness essentially as attribute completeness. Every object must have at least one attribute. Any attribute that is defined must be used in at least one expression (or provided by input). These criteria are easily evaluated by the Model Generator and are not seen as an intrusion into the domain of the Model Analyzer.

4.4 Database Design

The Model Generator captures model definition, specification, and documentation and stores these as relations in an INGRES relational database. The Model Generator utilizes two types of relational databases. The first database, **mgmodels,** contains a single relation which holds the name of all models under development as well as sponsor and modeler information for each model. Each

model under development has its own database, with the same name as the model, which contains the model definition, specification, and documentation.

The Model Generator can access the database utilites **makedb, initdb, loaddb, cleandb,** and **killmods.** Each utility is described extensively in its program documentation and therefore an in-depth discussion here is omitted. Briefly, makedb creates the database given by the model name and creates the relations that will hold the model information. Initdb loads the database with some model information that is universal such as adding the action STOP to the condition Termination, and attaching the temporal transitional indicative attribute *system_time* to the top level model object. (Note: the Model Generator assumes the indexing attribute of the model is time, as is typical in discrete event simulations, and not space or some other dimension.) Loaddb loads a database with the values for the machine repairman problem to facilitate experimentation. Cleandb removes all data values from relations in a model database, and killmods interactively destroys model databases.

A model database includes relations for model documentation such as model objectives, assumptions, and definitions, relations for model objects and attributes, relations for each condition and action type, as well as relations which link conditions and actions to represent the model CAPs. These relations are described in detail in the program documentation for makedb.

4.5 Interface Design

Since the Model Generator is first and foremost an experimental research prototype designed under the evolutionary and rapid prototyping paradigms, many user interface issues considered vital to a marketable product are neglected in the Model Generator interface. The primary focus of this Model Generator prototype is to provide a platform for the analysis of the Condition Specification as the specification form in the SMDE. Accordingly, this research does not claim to be in the area

of user interfaces, and the Model Generator's interface should in no way be mistaken as an example of an ideal interface. However, in order to adequately assess the CS, the user interface must not be a hinderance. Therefore, the Model Generator interface is designed with attention given to the principles identified in Section 2.3. Admittedly, where the principles of rapid or evolutionary prototyping and user interface design clash, the Model Generator favors the prototyping paradigm. For example, the rapid prototyping paradigm states that code should be written to expect only valid input and ignore the possibility of invalid and erroneous input. User interface design espouses the identification of all possible errors and the provision of descriptive error diagnostics to the user. The Model Generator, although recognizing some typical and important modeler errors, does not check for bad input on most occasions.

4.5.1 Model Generator Interface Specification Using User Action Notation

Some major aspects of the Model Generator interface are designed using User Action Notation (UAN) [Hix and Siochi 1989]. The purpose of this is two-fold. Firstly, it provides a degree of much needed structure to the design of the Model Generator interface, and secondly since UAN is in its early stages, this effort supplies the authors of UAN with another test case and, hopefully, some valuable feedback on the technique.

UAN is a collection of representation techniques that are used to describe asynchronous, event-based, highly complex and dynamic direct manipulation interfaces [Hix and Siochi 1989]. This scheme depicts the behavioral aspects of human-computer interfaces, i.e. they reflect a view of the system from a user's perspective. UAN represents a potentially powerful arena for interface specification that permits communication between system designers and system builders, and most importantly perhaps, between system designers and system procurers. The collection of techniques identified by [Hix and Siochi 1989] consists of:

- User Action Notation (UAN) - a notation for describing:
  - User actions: what the user phisically does while using the interface,

- Interface feedback: system responses cooresponding to the user actions, and
- System state: internal state changes that occur in response to the user actions.

- Scenarios - screen pictures that complement the UAN descriptions.
- Transition diagrams - state diagrams representing structural flow of control among interface screens and states, most useful when interaction among tasks is conditional and complex.
- Annotation - notes appended to UAN descriptions, scenarios, etc., as needed for clarity.
- Discussion sheets - information concerning design questions, trade-offs, and decisions.

Table 6 presents a summary of UAN notation. An example of UAN, given in Figure 23, describes the system behavior of an Apple Macintosh when deleting a file from a desktop. The UAN descriptions of the Model Generator interface are given in Appendix A.

4.5.2 Exploitation of UNIX Multitasking

The large screen display and multiwindow display manager provided by the SUN environment allow the Model Generator to take advantage of the multitasking capabilities of the UNIX operating system. Within a "dumb terminal" interface, spawned processes must generally run in the background. In the SUN environment, however, one running process may spawn another process in a separate window; thus allowing a user to observe and interact with both running processes within the same display.

A Model Generator might utilize multitasking in a variety of ways. For example, the phases of the CM may exist separately as independent (but cooperating) processes, reinforcing in the mind of the modeler the phasal approach to model development under the CM. The implementation of this concept is realized in an early version of the current prototype. However, highly complex interprocess comunication required to capture the nature of the synchronization between the two phases, as well as a conservative locking mechanism under SunINGRES, combine to hamper the early prototype's usefulness for experimentation. The current version, therefore, manifests defi-

Table 6. Summary of UAN Notation

[Hix and Siochi 1989]

USER ACTIONS	
~	move the cursor
[sym]	context of sym; "handle" with which cursor manipulates sym
[x,y]	context of screen location x,y
X v	depress button X
X ∧	release button X
()	grouping mechanism
*	kleene star; indicates zero or more repetitions of previous action
&	concurrency symbol; used to indicate that actions it connects are performed together, but are order independent
;	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 action)
FEEDBACK	
!	highlight an object
-!	dehighlight an object
!!	same as !, but use an alternate highlight
CONDITIONS OF VIABILITY	
conditon : action	
	if condition is true, then action may be performed, e.g., X-!: ~[X]Mv∧ means "if X is not highlighted, user may click the mouse on it." An alternative form is ~[X-!] M:v∧, which means "move to an unhighlighted X and click on it."

TASK: Deleting a Macintosh File		
User Actions	**Feedback**	**System State**
~[file_icon -!]; Mv	file_icon!	file_icon is selected
~[x,y]*	outline of file_icon follows cursor	
~[trash_icon];	trash_icon!	
M∧	erase file_icon, trash_icon!!	mark file for deletion

Figure 23. UAN Example -- Deleting a Macintosh File
[Hix and Siochi 1989]

nition and specification within the same process. Multitasking is heavily utilized in the prototype, for example, the Model Generator allows the interactive editing of model assumptions/objectives/definitions in one window, listings of model functions, and object attributes in other windows, and at the same time providing definition and specification in another.

Multitasking is exploited in the SMDE as a whole. As shown in Figure 24, the tools of the SMDE are activated as spawned processes. A particularly attractive feature of multitasking within the SMDE is that on-line help in the form of the Assistance Manager [Frankel and Balci 1989] can run simultaneously with the tool for which help is desired. The number of processes running at the same time in an SMDE session is limited by the SUN environment to approximately 60.

4.5.3 Advantages of Iconic Interfaces

The SUN-3/160 provides a 3-button multifunction mouse which allows user interface designers to stress iconic selection input. Iconic interfaces are attractive to users because they minimize effort and provide enhanced visual feedback. Early prototypes are keyboard based and require, by today's standards, excessive text entry. The Model Generator provides a "point and click" interface requiring text entry only when user responses cannot be predetermined from possibilities, e.g. when entering model documentation.

4.5.4 Explicit Display of Model Hierarchy

For large modeling efforts in particular, a modeler has a critical need to be aware of the hierarchical context of the current development effort. The Model Generator is driven by a model hierarchy display as shown in Figure 25. The scrollable panel to the left of the definition/specification driver contains the model hierarchy. The current object is highlighted in the list and displayed in the

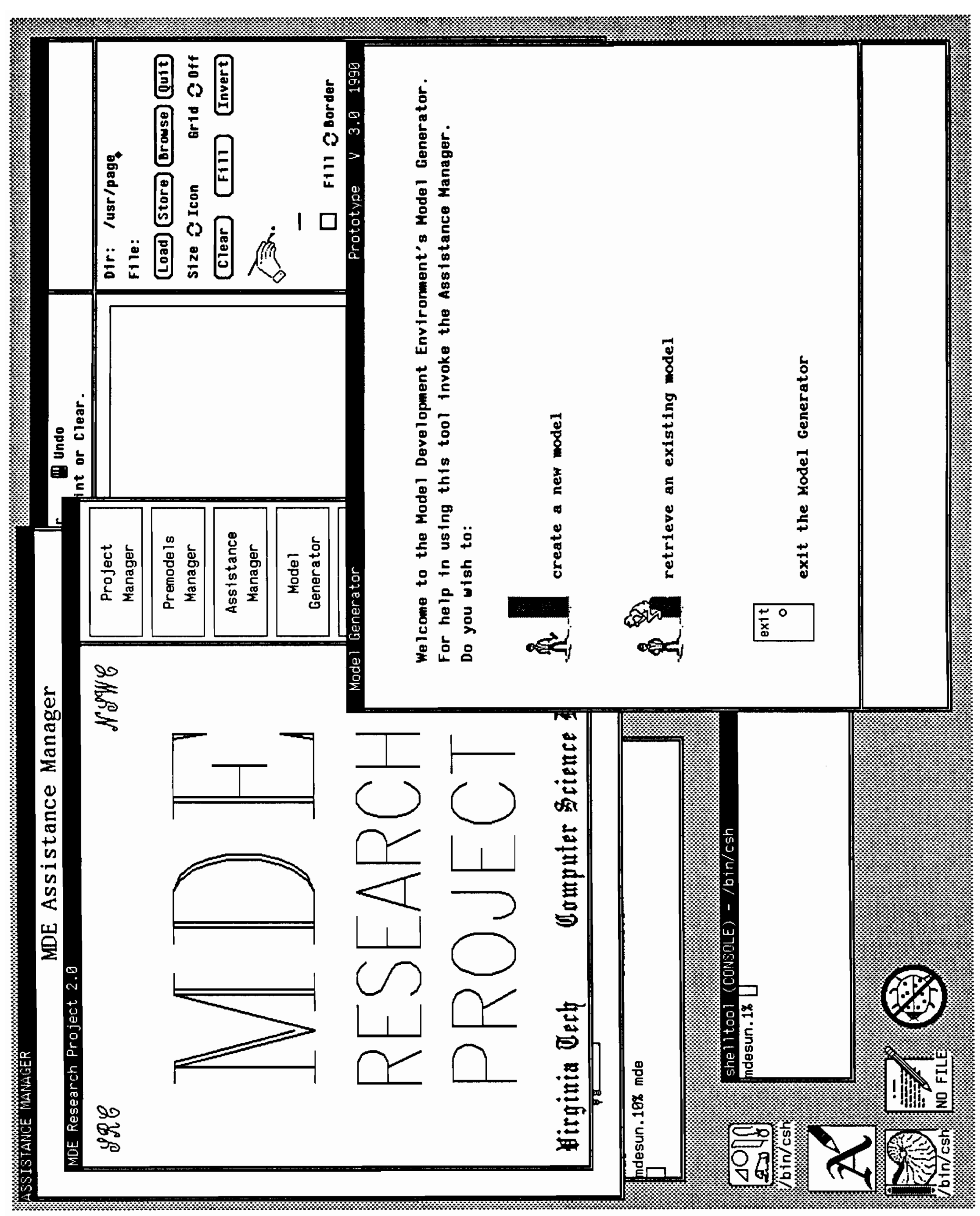

Figure 24. The Model Generator in the Simulation Model Development Environment.

window label as well. If the model grows too large, the window may be resized by the user during interaction with the Model Generator. If the model hierarchy becomes larger than the screen, the scrollable panel allows a modeler to locate the current object and display its context to as large (or as small) a degree as possible within the constraints of the SUN console monitor.

4.5.5 Rapid Shift Between Definition and Specification

The CM advocates phasal development of a model in a cyclic manner. The modeler is expected to shift between definition and specification often. Therefore, the Model Generator should accomodate this as simply as possible.

As mentioned in section 4.5.2 an early version of the current prototype implements model definition and model specification as two separate processes. The primary motivation for this configuration is to (1) enforce phasal model development and (2) provide the ability to shift rapidly between phases. This design, although not utilized in the current version of the prototype due to the system limitations discussed earlier, may be realized in future prototypes. The Model Generator allows the transition from definition to specification by requiring no more than two selections to arrive at the definition/specification driver (shown in Figure 24) from most points in the development process. While not *ideal,* shifting between phases is sufficiently rapid so as (hopefully) not to burden the user.

4.5.6 Automatic Revision for Configuration Management

The software engineering community recognizes the issue of configuration management as a problem that must be addressed. In large software projects, where many groups may be working on different portions of the same piece of software, groups are forced to operate on identical and cur-

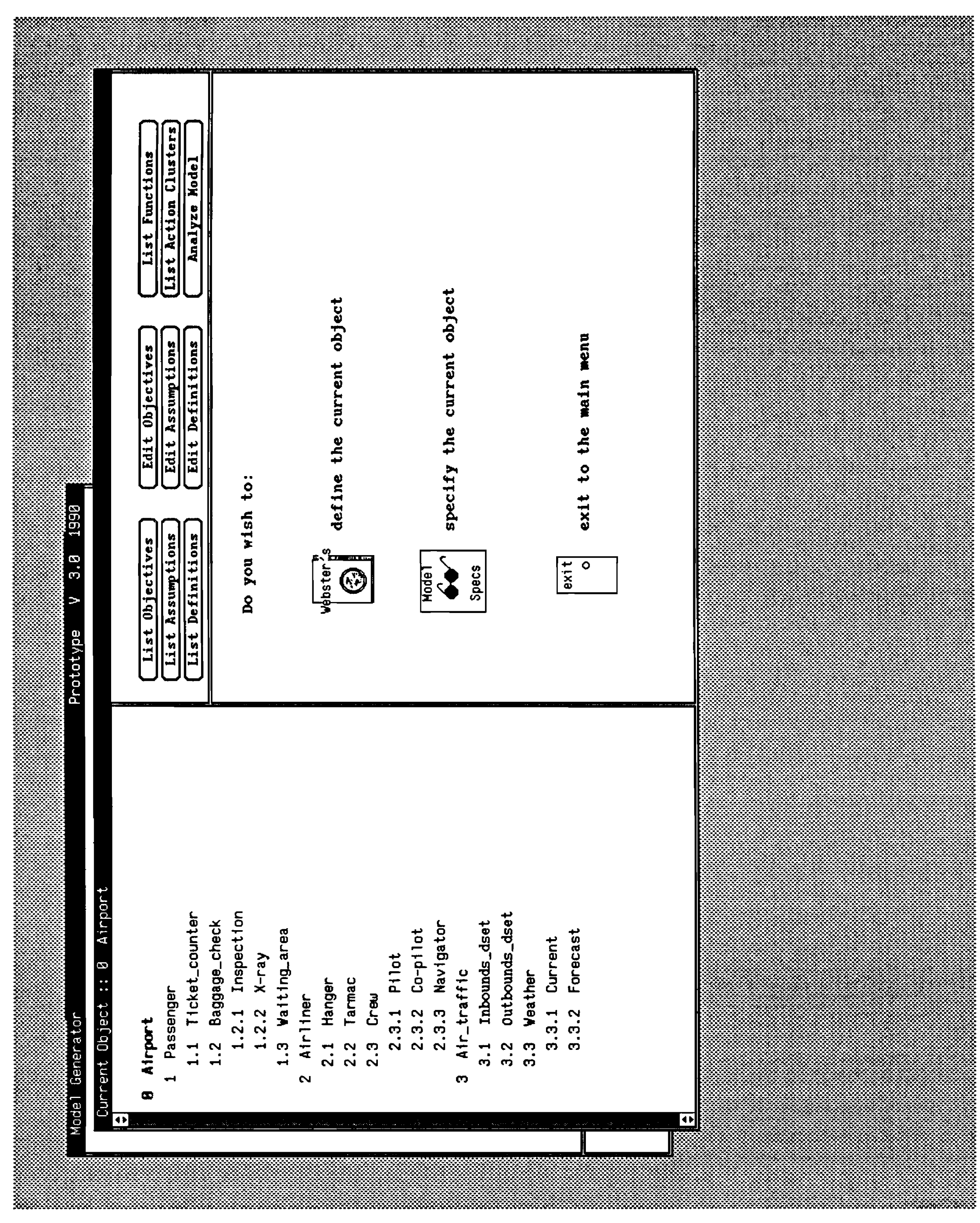

Figure 25. The Model Hierarchy Display.

rent versions of shared routines. Simulation models are some of the largest software applications in use today. The SMDE provides the Project Manager and Premodels Manager facilities which aid in the manipulation and selection of existing submodels that can be brought together to form models in the Model Generator. This system seeks to exploit another key software engineering objective, that of component reuse. When an existing component is modified within the Model Generator, the change is noted. Within the framework of a fully integrated SMDE, this information could be used to identify which component versions are being utilized by various model development groups working on the same, or even different, simulation modeling efforts.

4.6 Summary

The Model Generator, its documentation, and associated utilities consist of approximately 20,000 lines of code in C and EQUEL-C. The Model Generator represents an example of evolutionary prototyping in that it builds upon the *concepts* developed by [Hansen 1984] and [Barger 1986], although no Hansen or Barger code could be reused.

The Model Generator captures a complete implementation of the Condition Specification, providing the specification of attributes, model input and output, object creation and destruction, model initialization and termination, and user defined functions. The Model Generator extends the capabilities of the CS to provide for set specification by defining set manipulation primitives.

The Model Generator utilizes relational database technology to capture model specifications, and the Model Generator interface is designed to take advantage of the capabilities of the SUN windowing system. An analysis of the Model Generator design based on the requirements of the prototype as well as identified properties of "good" specifications is presented in Chapter 6.

5 Example

This chapter presents *a few* aspects of the Model Generator as evidenced in the development of the machine repairman model. A detailed description of *all* Model Generator utilities and functions is given in Appendix B.

The object specification and transition specification for the machine repairman problem produced by the Model Generator are given in Figures 26 and 27 respectively. This specification is equivalent to that given in Chapter 3 as well as those appearing in [Overstreet and Nance 1985; Nance and Overstreet 1988]; the only significant difference being the usage of dot-notation and the ownership of all time-based signals by the top level object. Also, no "idle position" object is created, since in the specification given in Chapter 3, the only attribute of idle position is a time-based signal. (Recall that the Model Generator, following the Conical Methodology, requires that every object have at least one attribute.)

Figures 28-53 illustrate selective features of model development using the Model Generator. The reader is advised to refer frequently to these illustrations while reading the narrative below.

Object	Attribute	Type
mrp	system_time	positive real
	n	positive integer constant
	mean_uptime	positive real constant
	mean_repairtime	positive real constant
	max_repairs	positive integer constant
	end_repair	time-based signal
	arr_fac	time-based signal
	arr_idle	time-based signal
	failure	time-based signal
facilities	fac	constant
	failed	boolean
repairman	status	(avail, travel, busy)
	location	(idle, fac)
	num_repairs	nonnegative integer

Figure 26. Model Generator Produced Machine Repairman Object Specification

```
{Initialization}
INITIALIZATION:
 INPUT(n,max_repairs,mean_uptime,mean_repairtime);
 CREATE(repairman);
 FOR i := 1 TO n DO
   CREATE(facilities);
   facilities.fac := i;
   facilities.failed := false;
   SET ALARM(failure, neg_exp(mean_uptime),i);
 END FOR
 repairman.num_repairs := 0;
 repairman.location := idle;
 repairman.status := avail;
{Termination}
repairman.num_repairs ≥ max_repairs:
 STOP
{Failure}
WHEN ALARM(failure,i):
 facilities(i).failed := true;
{Begin repair}
WHEN ALARM(arr_fac,i):
 SET ALARM(end_repair, neg_exp(mean_repairtime),i);
 repairman.status := busy;
 repairman.location := facilities(i).fac;
{End repair}
WHEN ALARM(end_repair,i):
 SET ALARM(failure, neg_exp(mean_uptime),i);
 facilities(i).failed := false;
 repairman.status := avail;
 repairman.num_repairs := repairman.num_repairs + 1;
{Travel to idle}
(FOR ALL 1 ≤ i ≤ n, NOT facilities(i).failed) &
repairman.status = avail & repairman.location ≠ idle:
 SET ALARM(arr_idle, traveltime(repairman.location, idle));
 repairman.status := travel;
{Arrive idle}
WHEN ALARM(arr_idle):
 repairman.status := avail;
 repairman.location := idle;
{Travel to facility}
repairman.status = avail & (FOR SOME 1 ≤ i ≤ n, facilities(i).failed):
 SET_ALARM(arr_fac, traveltime(repairman.location,
 closest_failed_fac(facilities.failed,repairman.location)),i);
 repairman.status := travel;
```

Figure 27. Model Generator Produced Machine Repairman Transition Specification

5.1 Initial Step in Model Definition

When creating a model, the modeler is requested to enter the top level model name. This name is used as the name of the database which stores all model documentation/definition/specification information. The name also appears at the top of the model hierarchy as the top level model object. The modeler is then requested to provide the name of the the model sponsor (for study and project documentation), the modeler's name, and the assumptions, objectives, and definitions to be utilized in the model. Figure 28 illustrates a model definition entry. Here, a modeler is providing a definition for an attribute of the top level object. Note that a modeler may also provide attribute descriptions when defining attributes (as will be illustrated later). After this information is entered, the model definition/specification driver is invoked. This is shown in Figure 29. Notice that the current object is indicated in the driver frame label as well as highlighted in the model hierarchy (initially, of course, the top level object is the *only* object). The phases of model generation are represented by the selectable icons labeled "define the current object" and "specify the current object."

The model assumptions/objectives/definitions are always available to the modeler throughout a session with the Model Generator. Selecting the button labeled "Edit Definitions" results in the selectable list of model definitions which may be edited as shown in Figure 30.

5.2 Defining Model Objects

Selecting the model definition icon shown in Figure 29 brings up the menu shown in Figure 31. The Model Generator allows a modeler to: "Attach" attributes to the current (selected) object; "Make" a subobject which will be directly subordinate to the current object; "Retrieve" a subobject

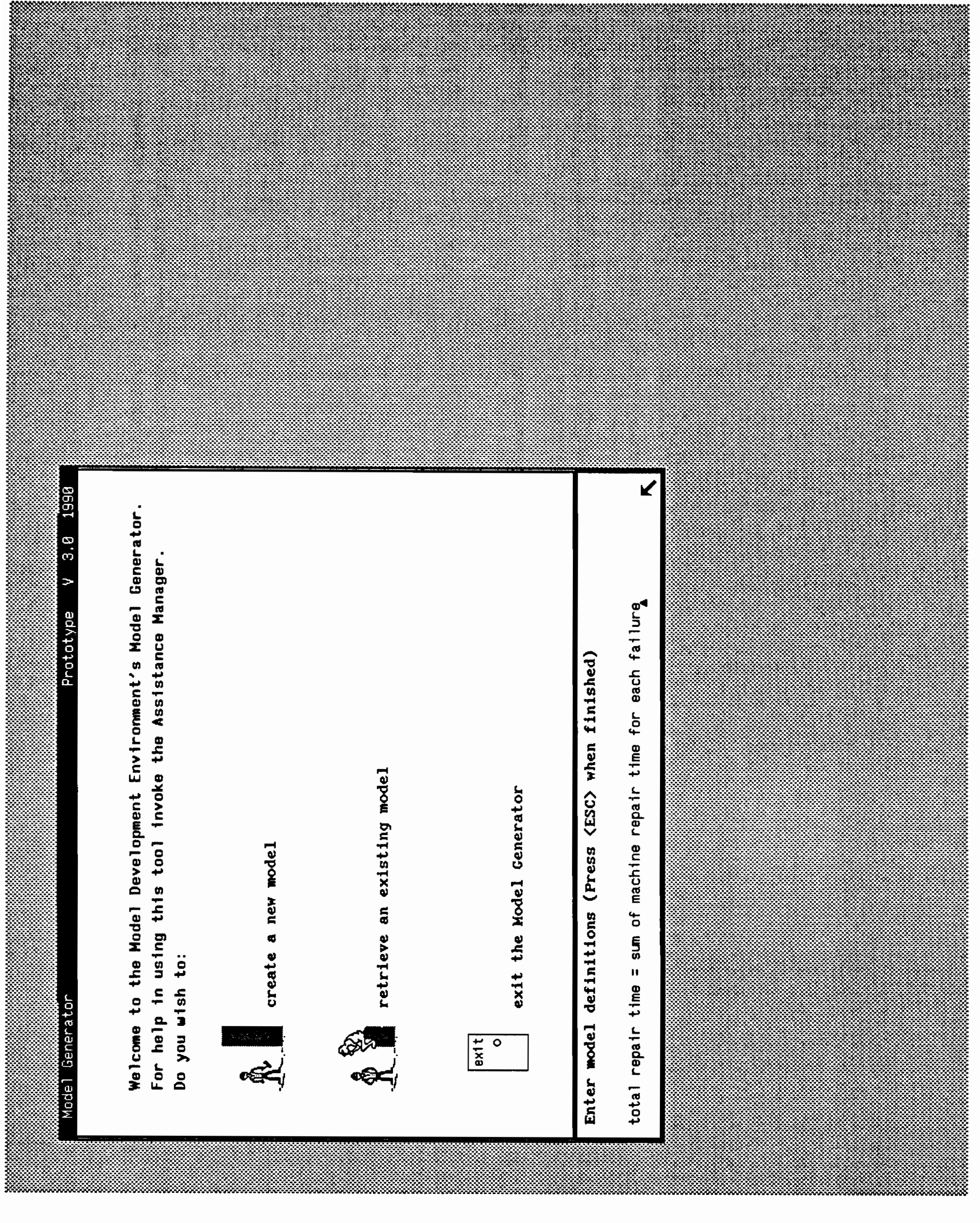

Figure 28. Entering Model Definitions.

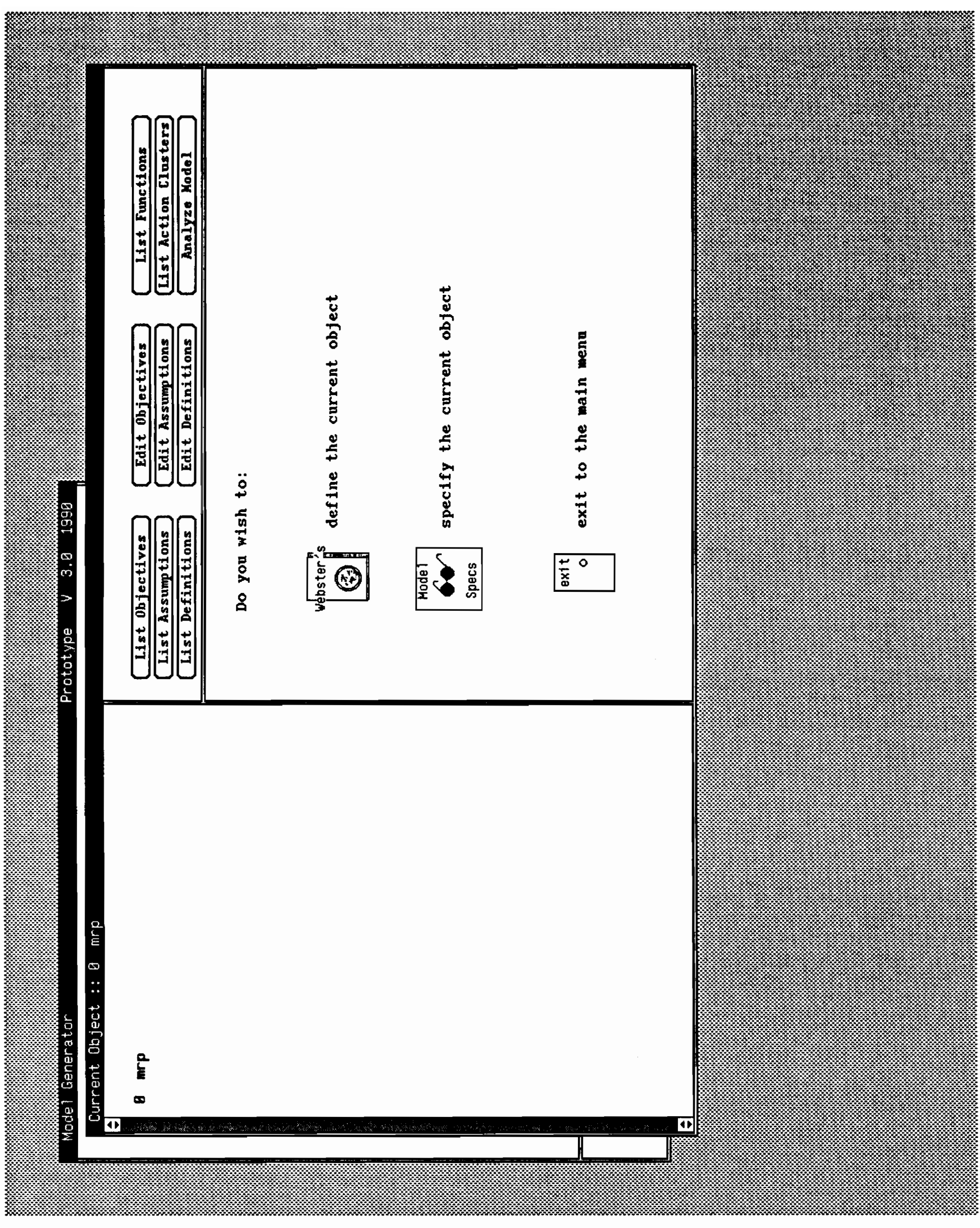

Figure 29. The Definition/Specification Driver.

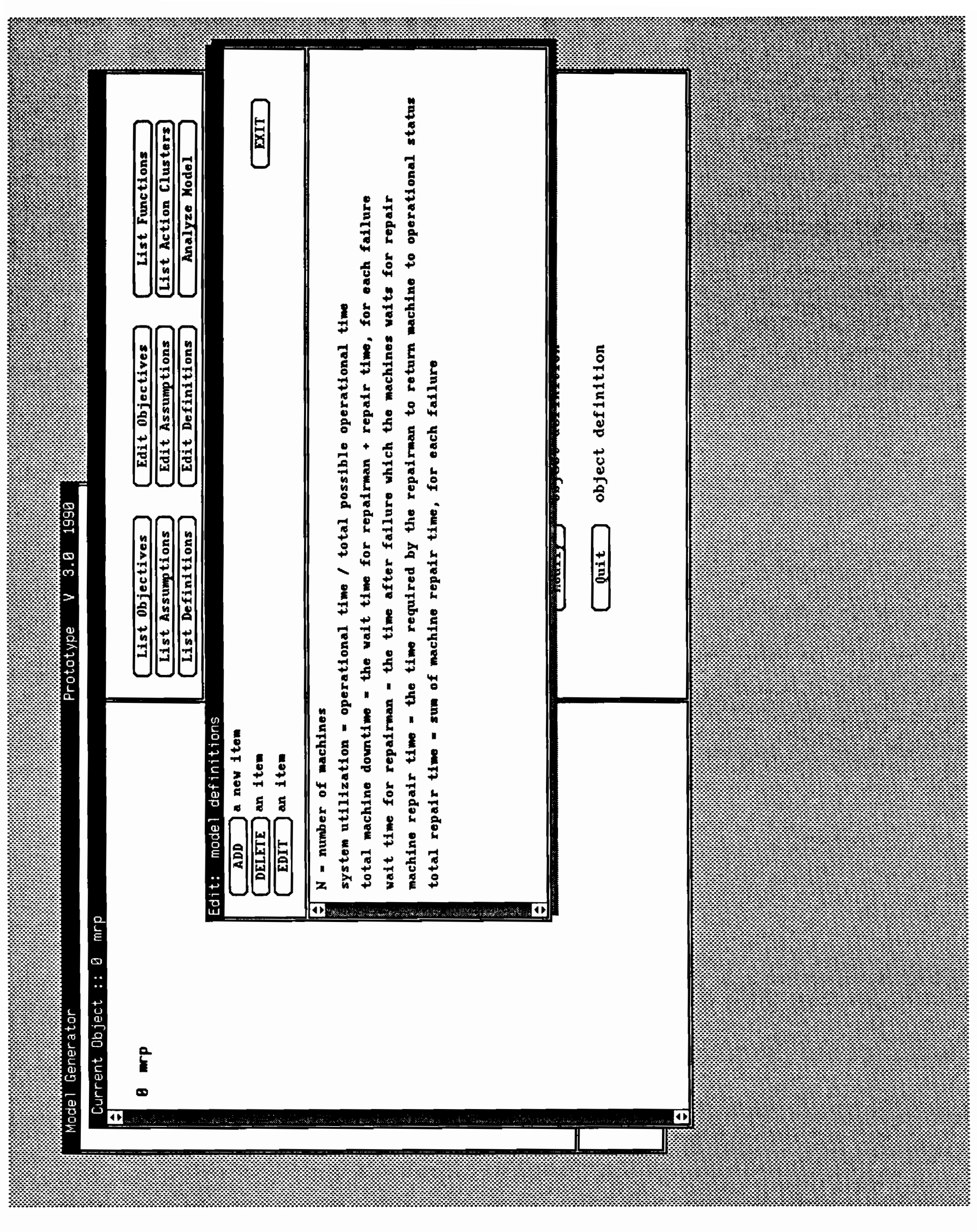

Figure 30. Editing Model Documentation.

from a model archive; "Create" a set which will be directly subordinate to the current object; "List" the defined attributes of the current object; "Delete" the current object; or "Modify" the object definition (either the object name or its attributes); Selecting "Make" results in the prompt shown in Figure 32. After entering an object, the model hierarchy is redisplayed with the new object as the current object. This is shown in Figure 33.

Selecting "Attach" from the model definition menu results in the menu displayed in Figure 34. The attribute must be named and a description of the attribute entered. Notice that the text of the attribute description scrolls to the left (as indicated by the dark triangle at the left of the description). This occurs because the amount of text that can be stored is larger than the amount that can be displayed. The CM type and dimensionality are also entered at this point. Clicking the left mouse button on the option (signified by the circling arrows) cycles through the possibilities one at a time. Clicking the right mouse button on the option raises a choice menu (shown for dimensionality) from which the desired option may be selected. Selecting from a menu may be preferable if the option list is long.

Selecting "list" from the model definition menu generates a list of attributes for the current object. The list of attributes for repairman is shown in Figure 35. Notice that the CS attribute types are at this time unknown since no model specification has taken place.

5.3 Model Specification

Selecting the model specification icon at the driver level (Figure 29) brings up the model specification menu shown in Figure 36. The Model Generator allows a modeler to specify: "Objects" of the model; "Initialization" conditions for the model; "Termination" conditions for the model;

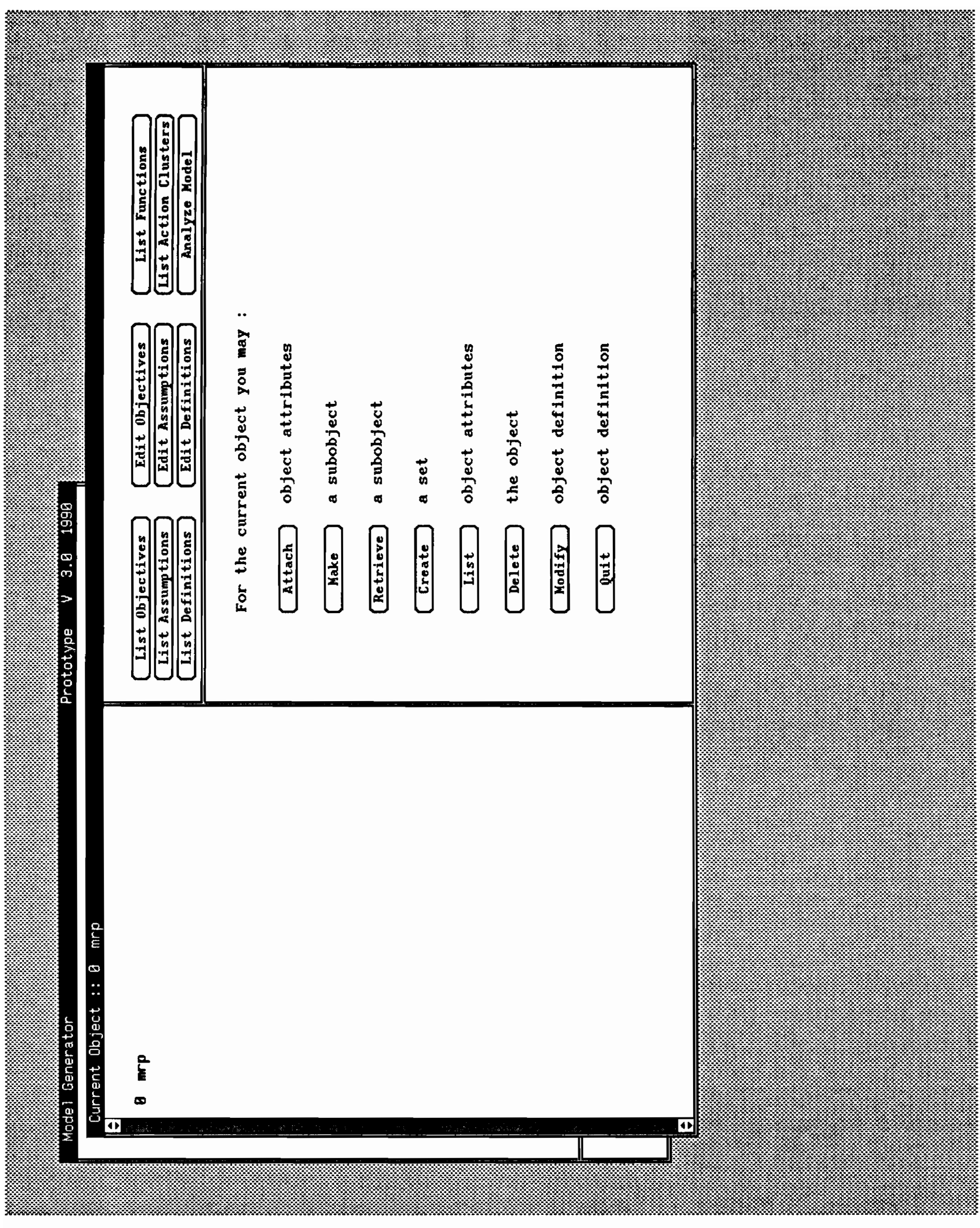

Figure 31. Model Definition Menu.

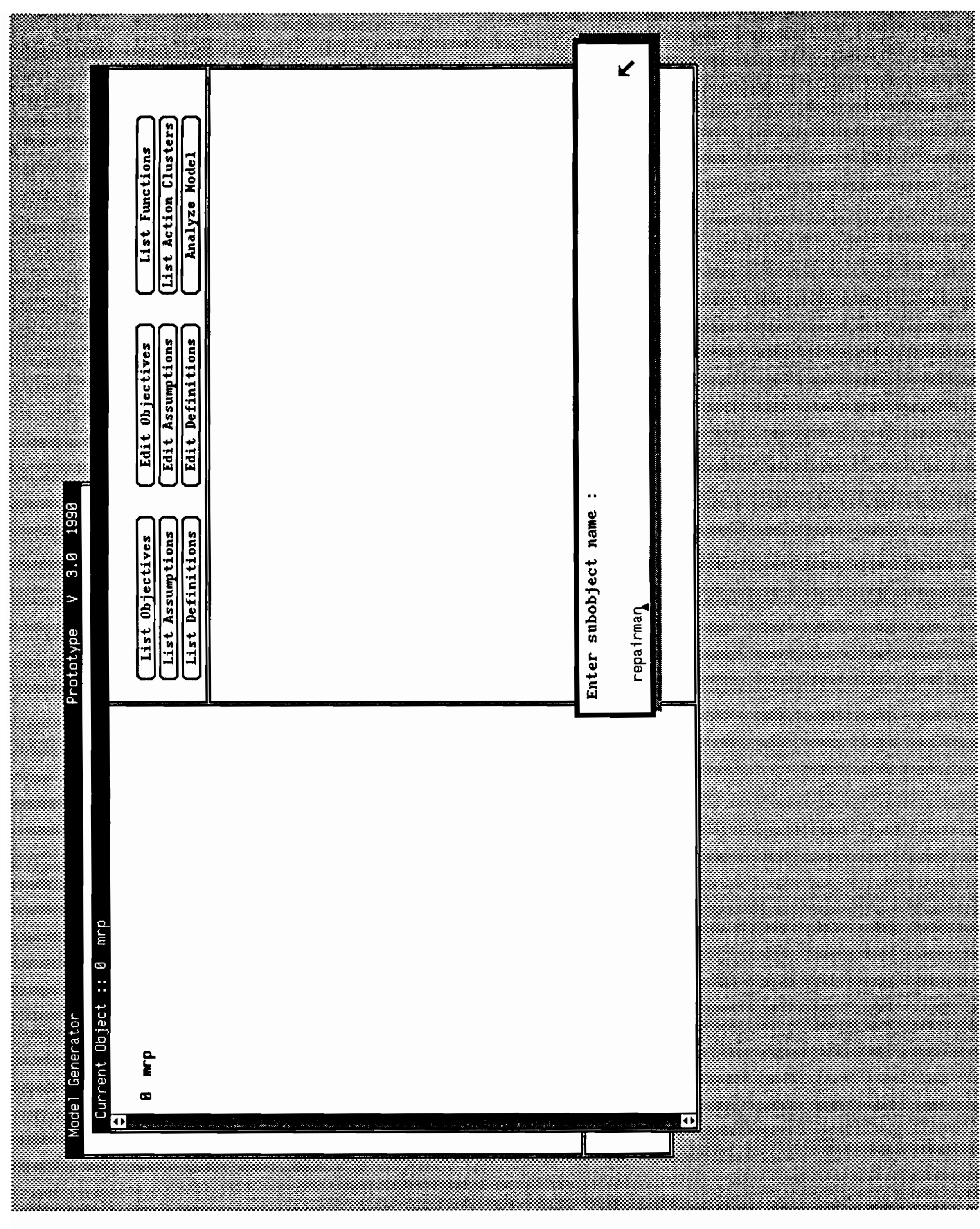

Figure 32. Entering New Model Object.

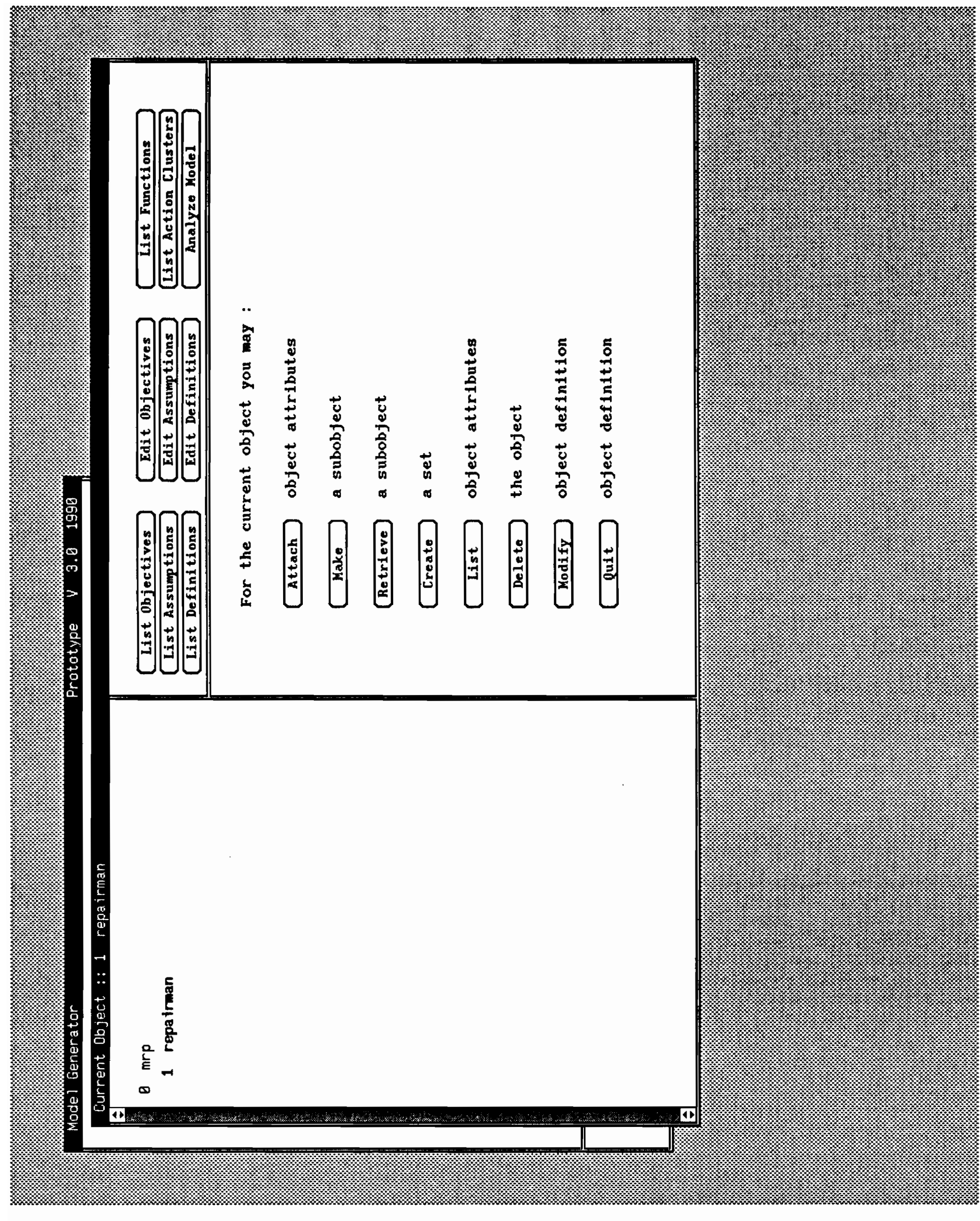

Figure 33. New Object Becomes Current Object.

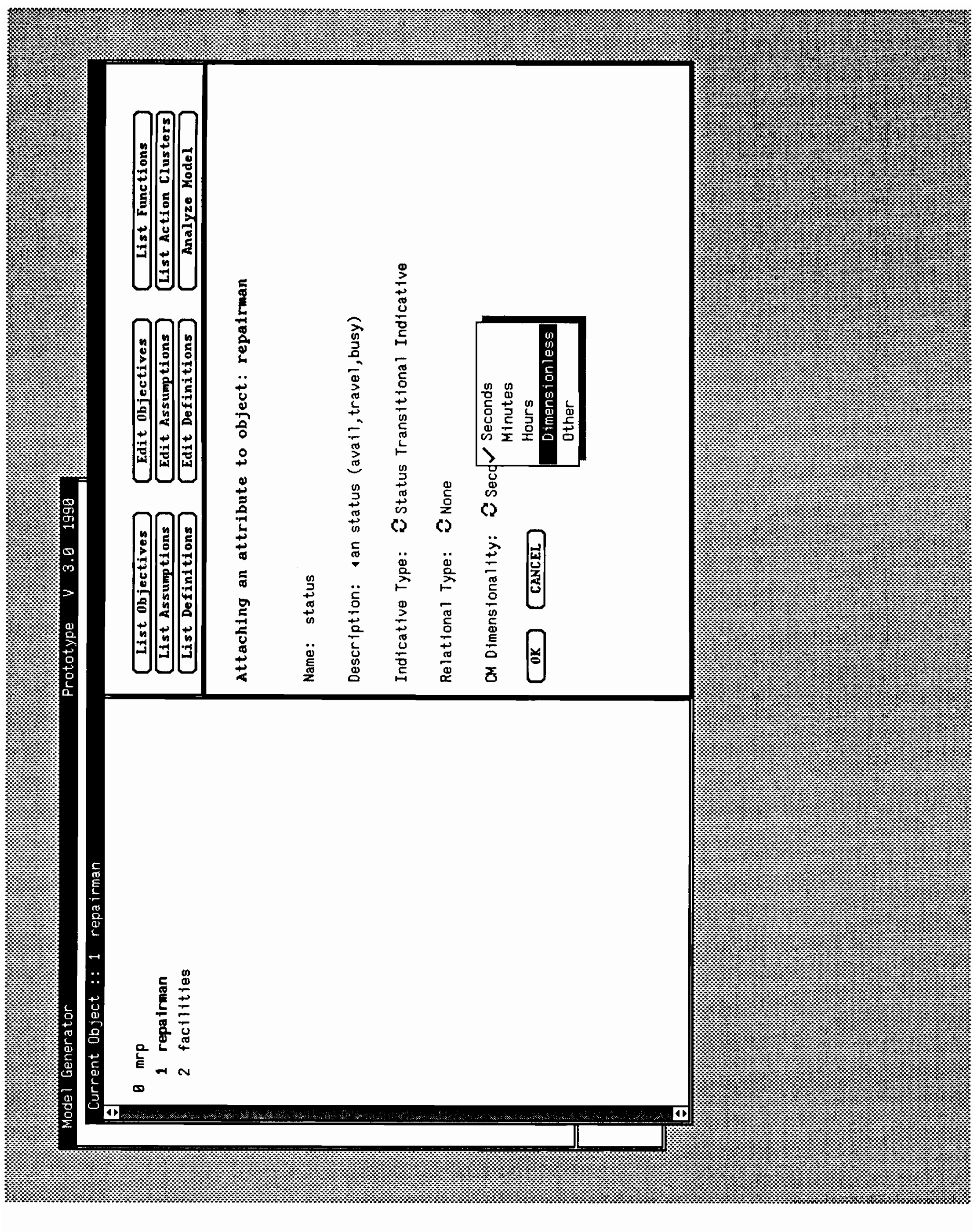

Figure 34. Attaching Object Attributes.

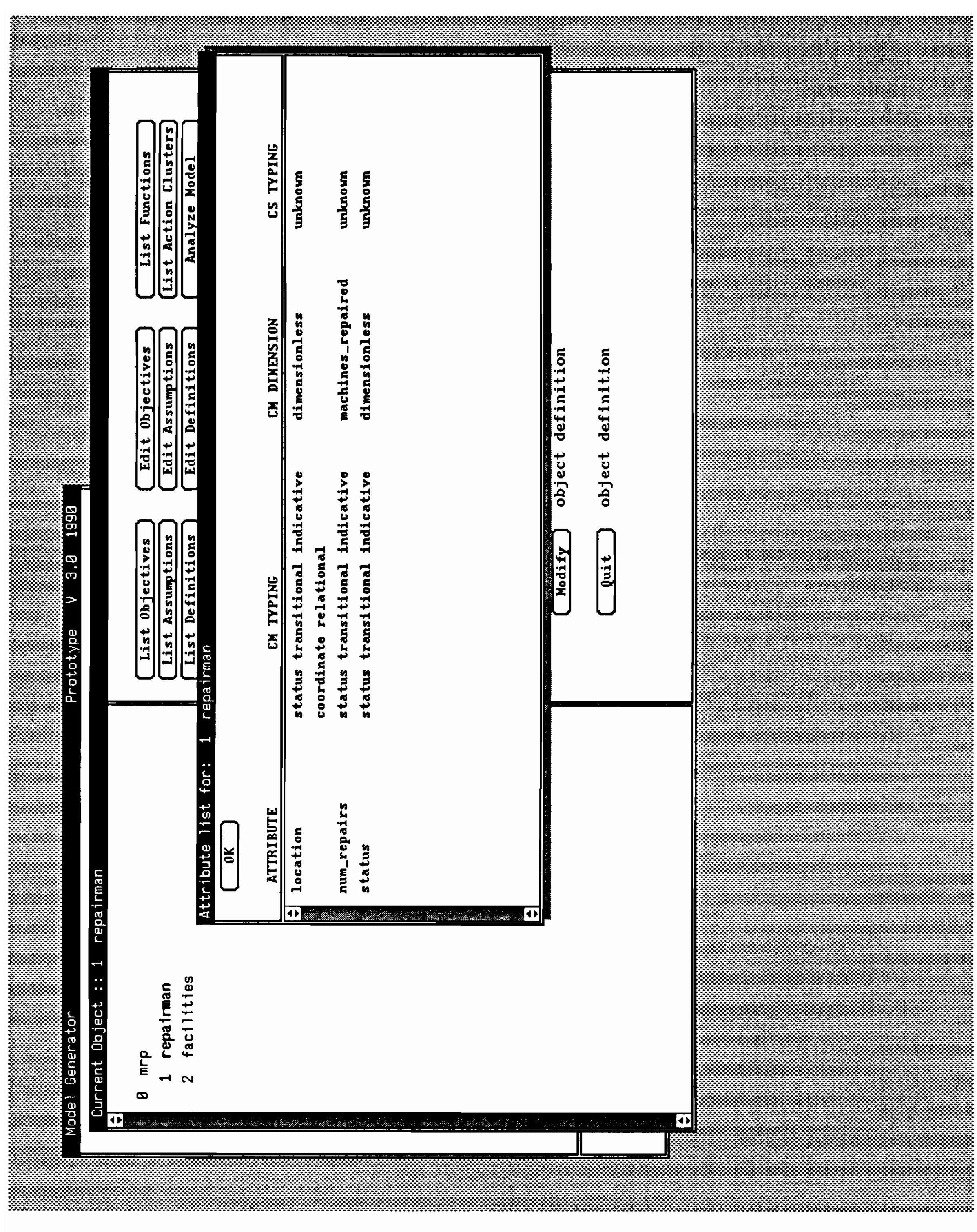

Figure 35. Listing Object Attributes.

"Input/Output" for the model; "Functions" referenced within the model; "Monitored" routines (attributes) within the model; or "Modify" model specification.

Selecting "Objects" brings up the object specification menu shown in Figure 37. Here, a modeler may specify: "Attributes" of the object; "Creation" of the object; "Destruction" of the object; or "Set" membership for the object.

Selecting "Attributes" from the object specification menu causes a selectable list of attributes of the current object to be displayed, as illustrated in Figure 38. Directing the modeler to select from a list of defined model attributes implicitly enforces the requirement that some definition take place prior to specification (at least for direct attribute specification). During attribute specification, if the CS type is not provided the modeler is warned as shown in Figure 39.

The modeler is requested to provide range or enumeration information for all attributes; this information is entered as shown in Figure 40.

5.3.1 Specifying Status Transitional Indicative Attributes

Status transitional indicative attributes are specified according to the conditions that cause them to change value. The menu for status transitional indicative attribute specification is shown in Figure 41. Value changes are added for sti attributes as illustrated in Figure 42. Figure 43 displays the complete list of value changes for the attribute *status* of the object ***repairman.***

Selecting "Specify" from the status attribute menu causes a value change requestor (similar to that in Figure 42) to come up and the value change to be specified is checked against those defined in the database. If the change in value exists, the value change condition menu shown in Figure 44 is displayed. If the change has not been defined, the modeler is warned and asked to re-enter the value change or cancel the current activity. To add a condition, a modeler may select an existing

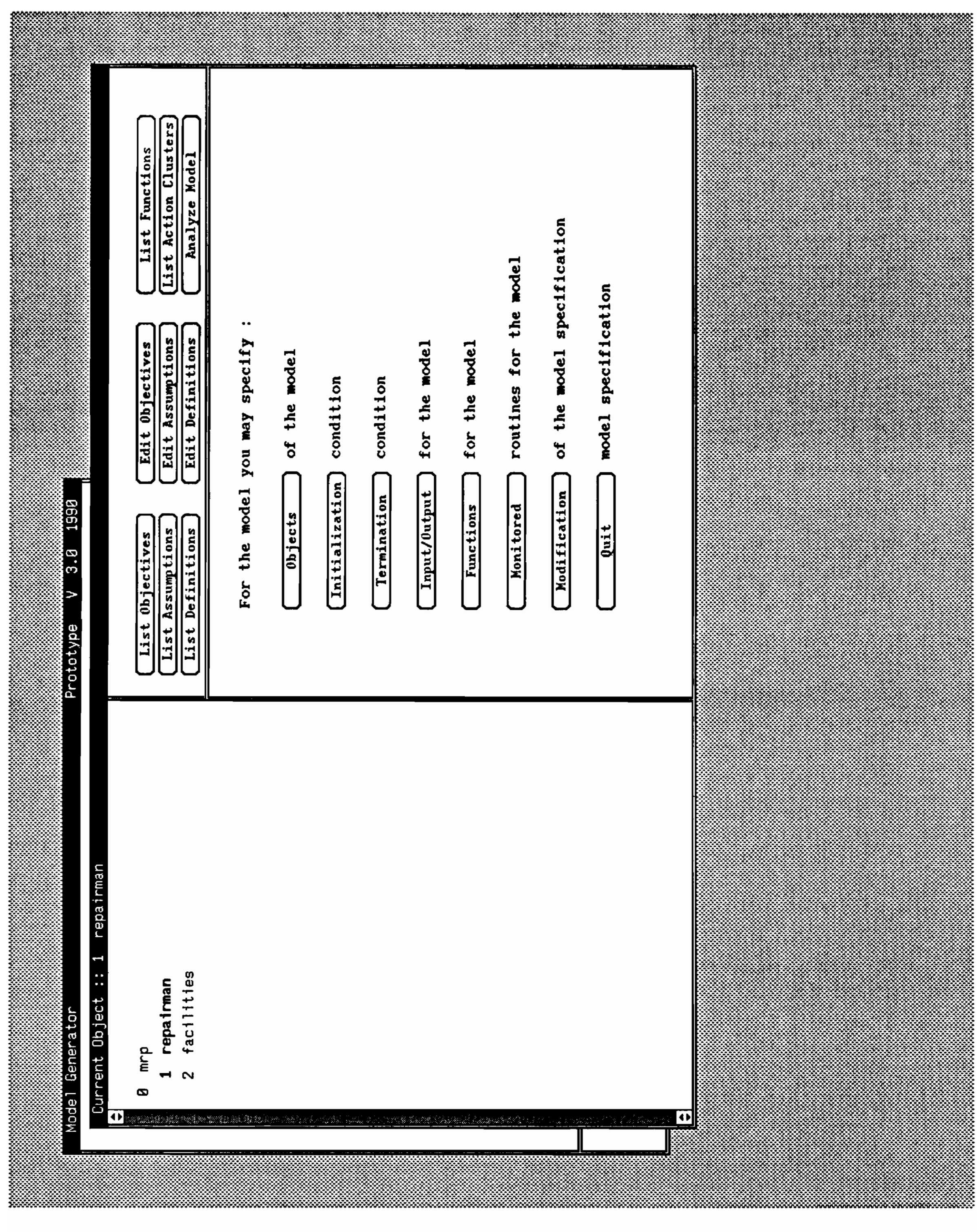

Figure 36. Model Specification Menu.

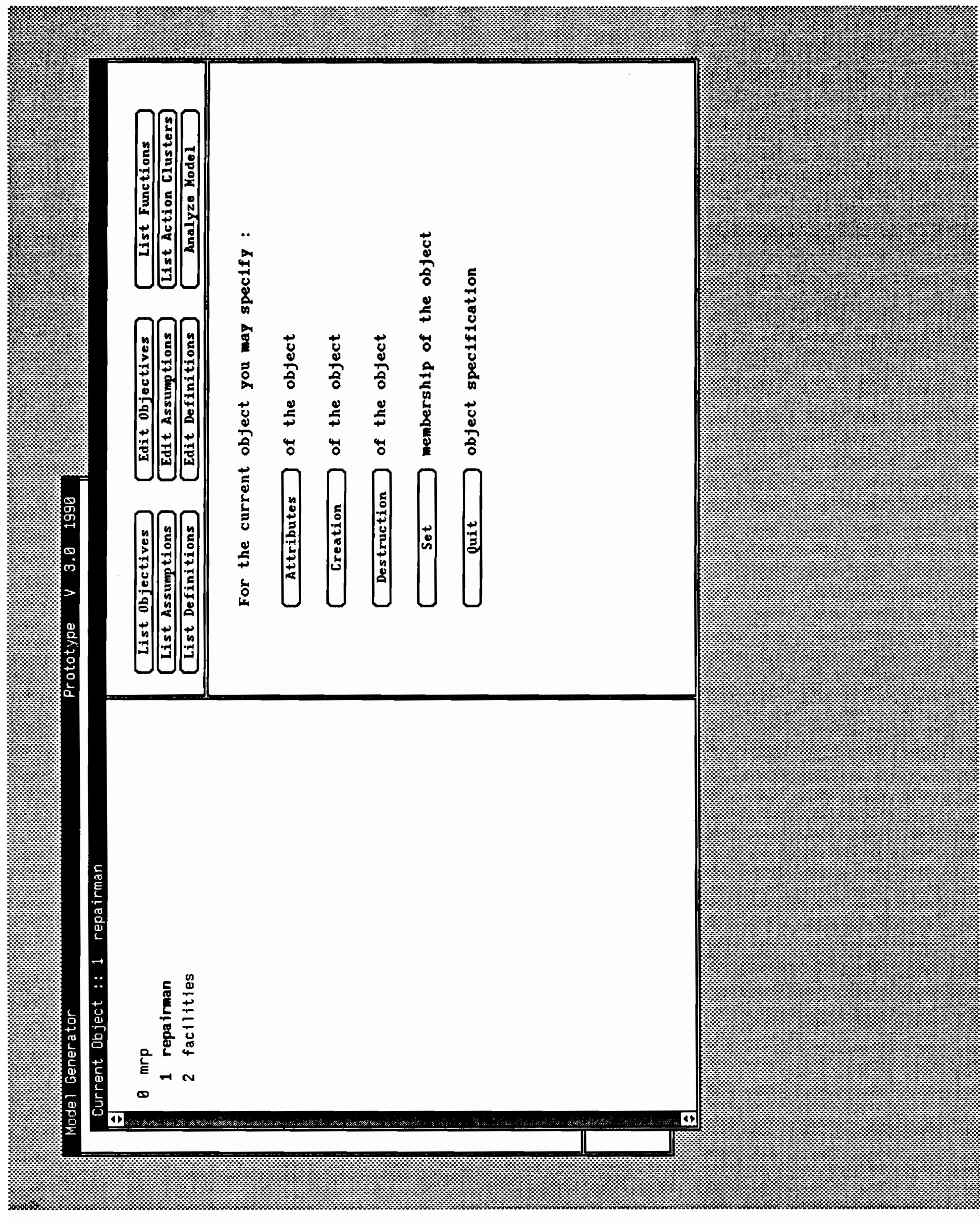

Figure 37. Object Specification Menu.

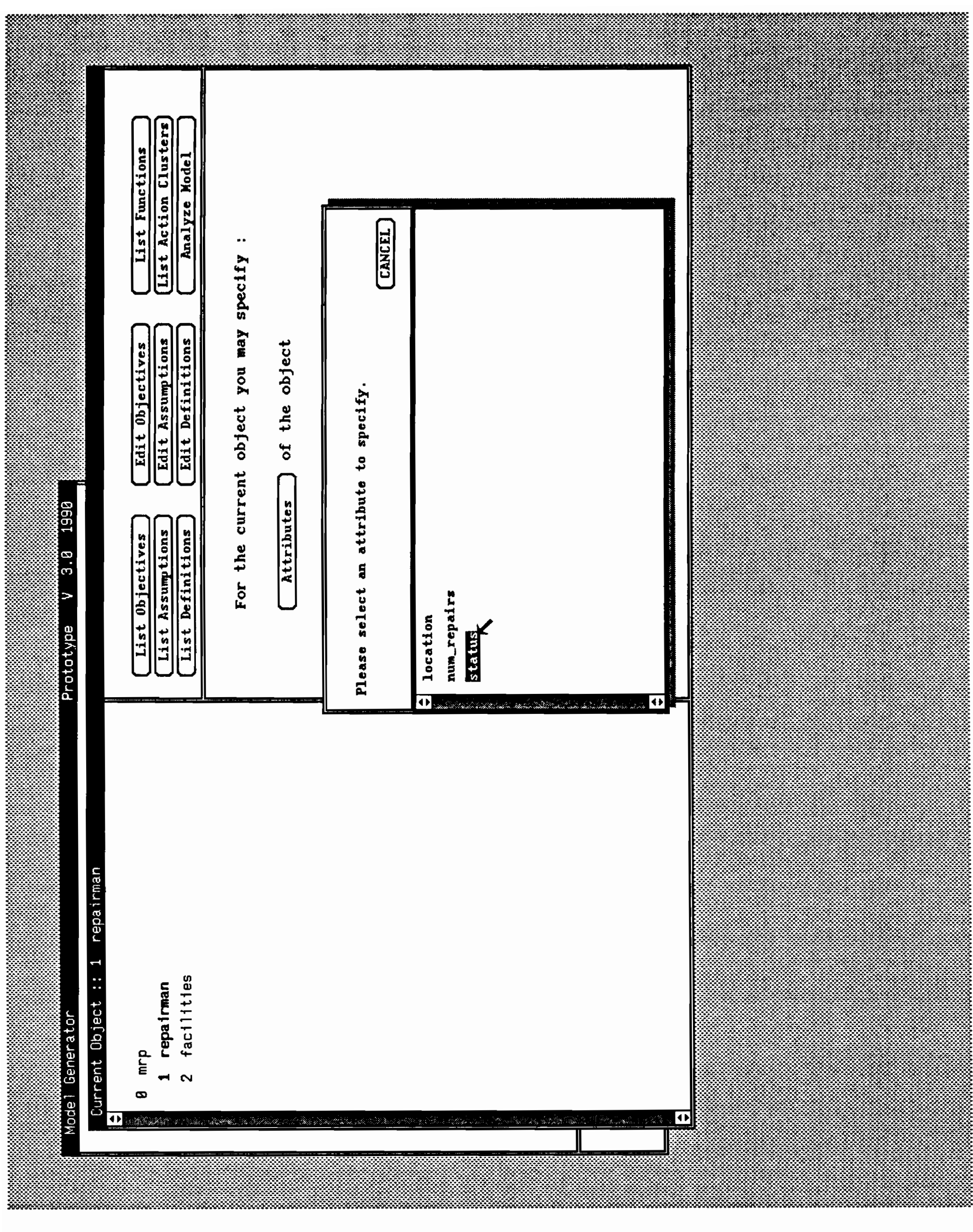

Figure 38. Selecting Attribute to Specify.

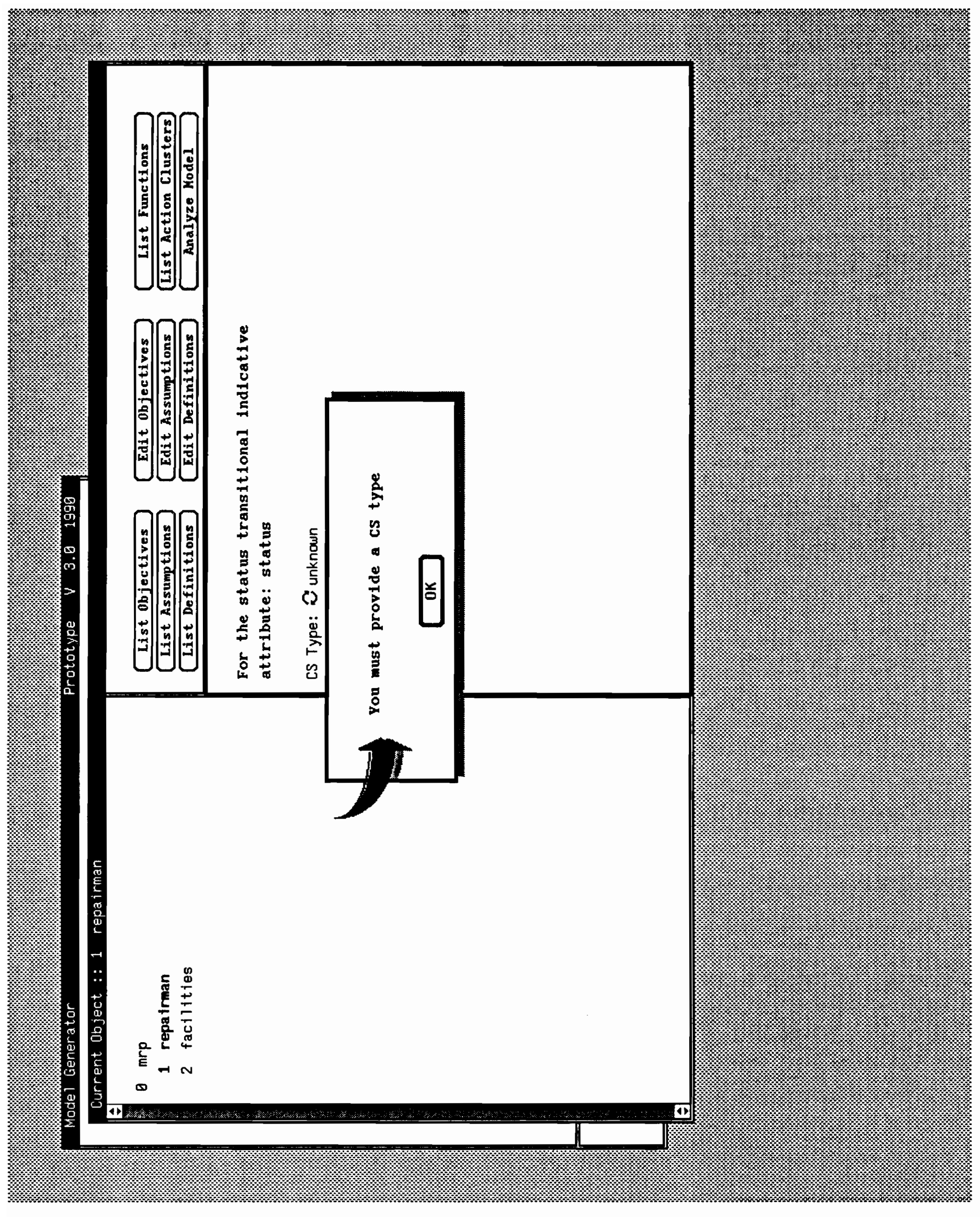

Figure 39. The Modeler Must Provide a CS Type for Attributes

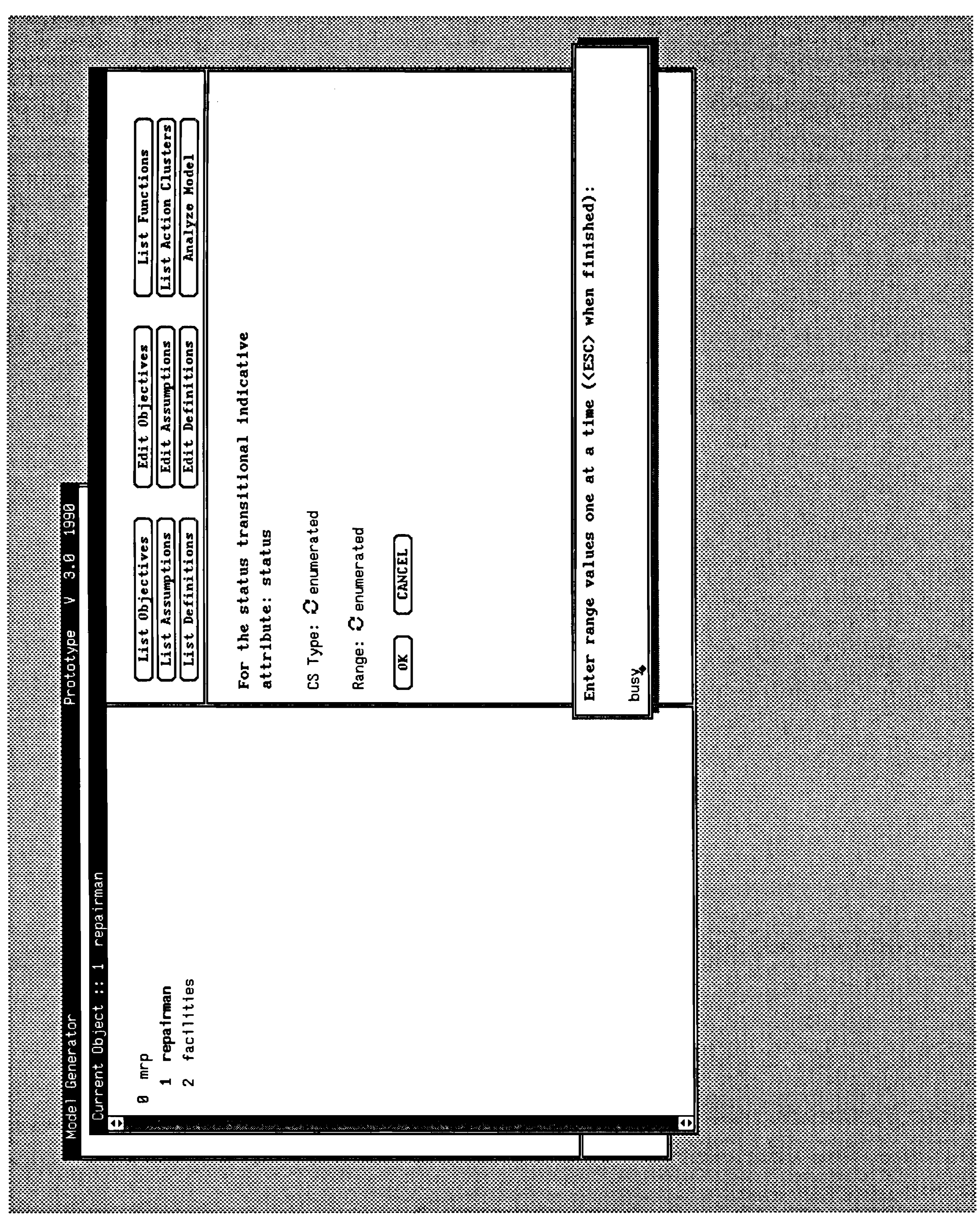

Figure 40. Enumerating the Values of Status Transitional Indicative Attributes.

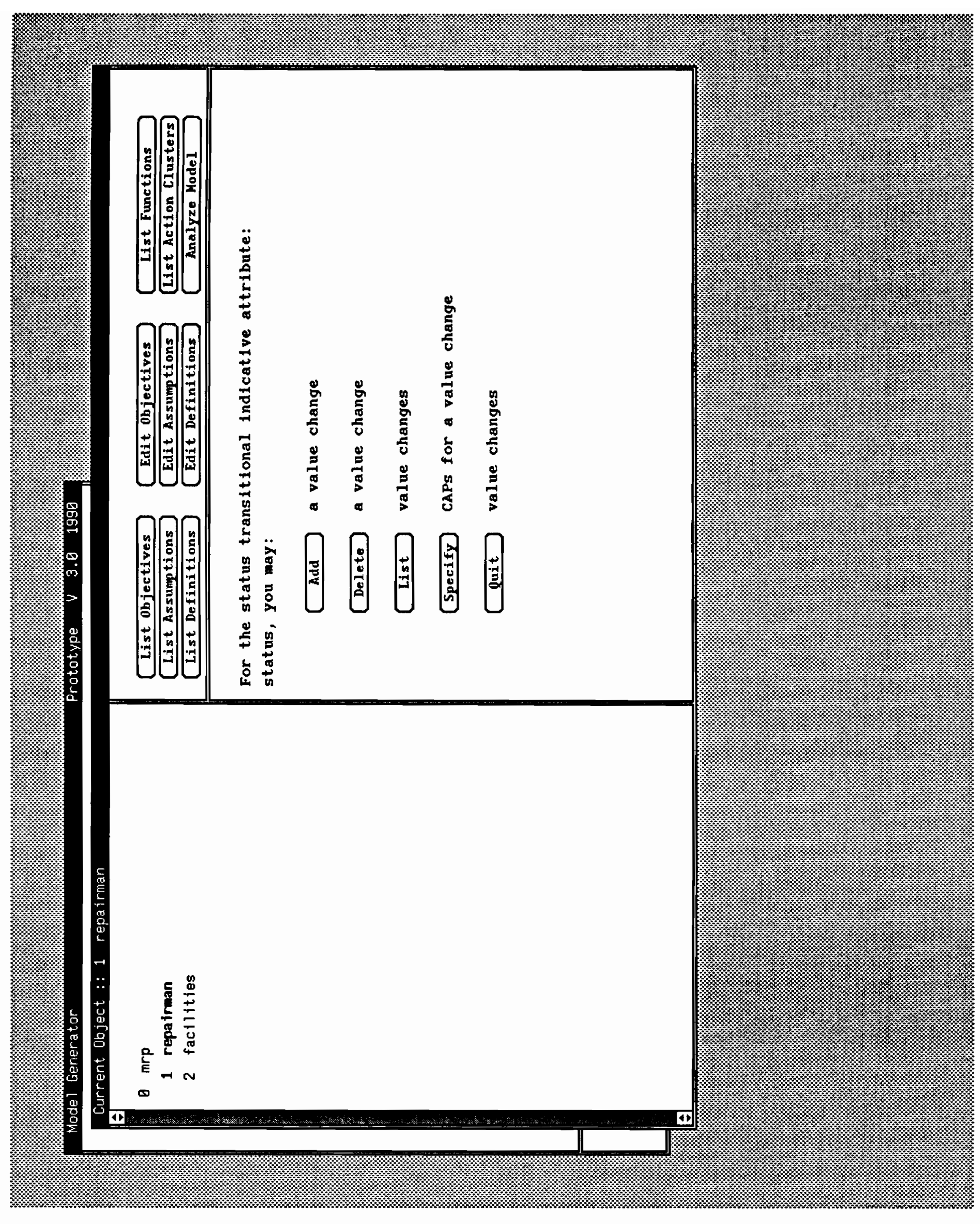

Figure 41. The Status Transitional Indicative Attribute Specification Menu.

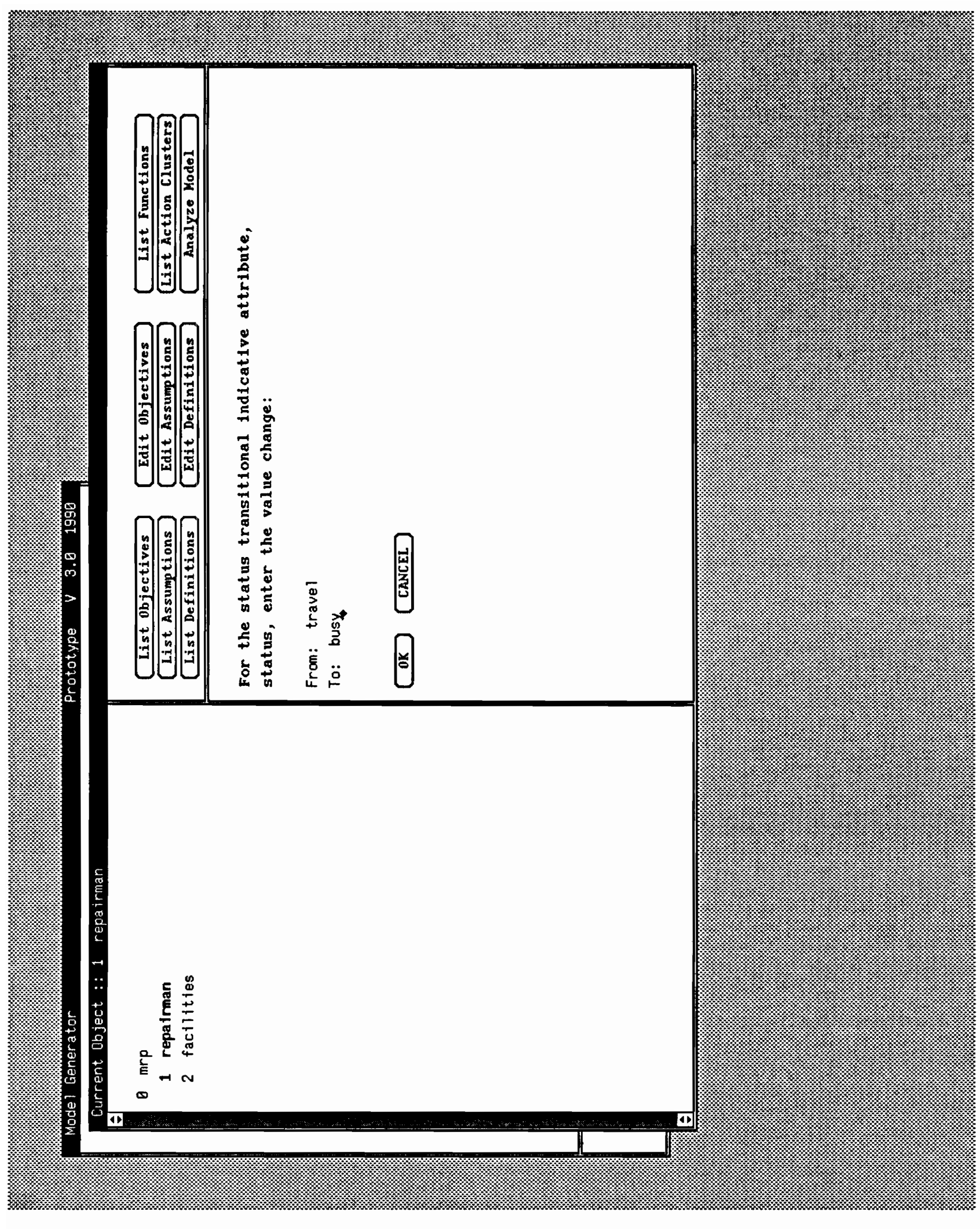

Figure 42. Adding Value Changes for Status Transitional Indicative Attributes.

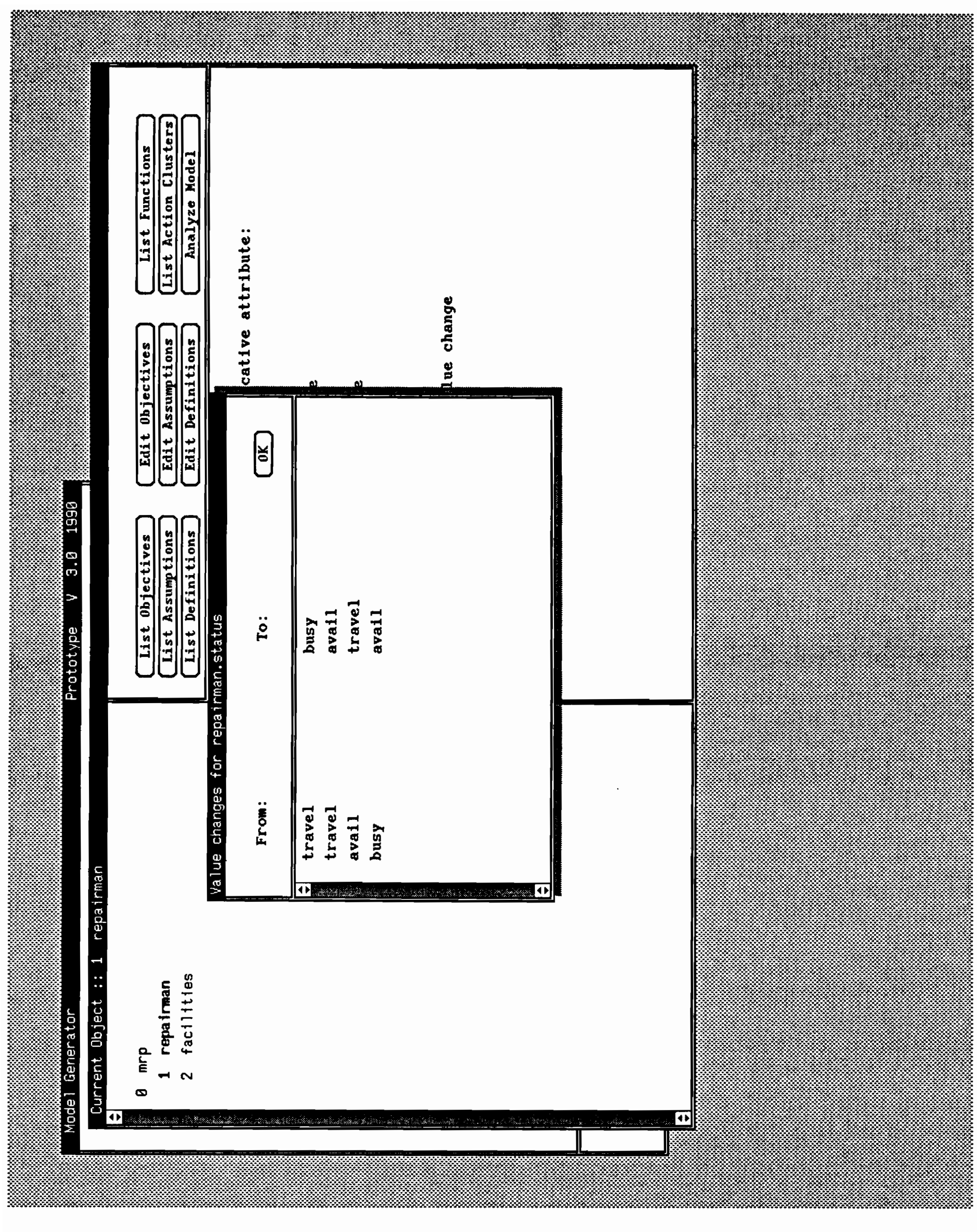

Figure 43. Listing Value Changes for Status Transitional Indicative Attributes.

condition from a list of model conditions or may define and specify a new model condition. Figures 45-48 show the addition of a new model condition, *travel_to_idle,* for the value change of the attribute *repairman.status* from *avail* to *travel.* When adding a condition, the condition name must be provided, as well as a description of the condition and the condition base (Figure 46). If the condition is state-based, a boolean expression for the condition must be provided (Figure 47). (Note that the text for the expression scrolls like the attribute description of Figure 34. Expressions are stored up to 500 characters, but the display length is necessarily shorter.) If the condition is time-based, an alarm must be provided rather than a boolean expression (see Section 5.3.2). If the condition is mixed, both a boolean expression and an alarm must be provided for the condition. When specifying conditions for sti attributes (Figure 44), a modeler may list the conditions causing a value change as illustrated in Figure 49. Here a modeler may view the entire text of condition expressions via the horizontally scrollable window.

5.3.2 Specifying Alarms

For conditions that are time-based or mixed, alarms must be provided. A modeler may select an existing alarm or provide a new one. Figures 50-52 show the specification of the time-based condition *arrive_idle.* Note that the new model alarm *arr_idle* is automatically attached as an attribute of the top level object as illustrated in the attribute list in Figure 53.

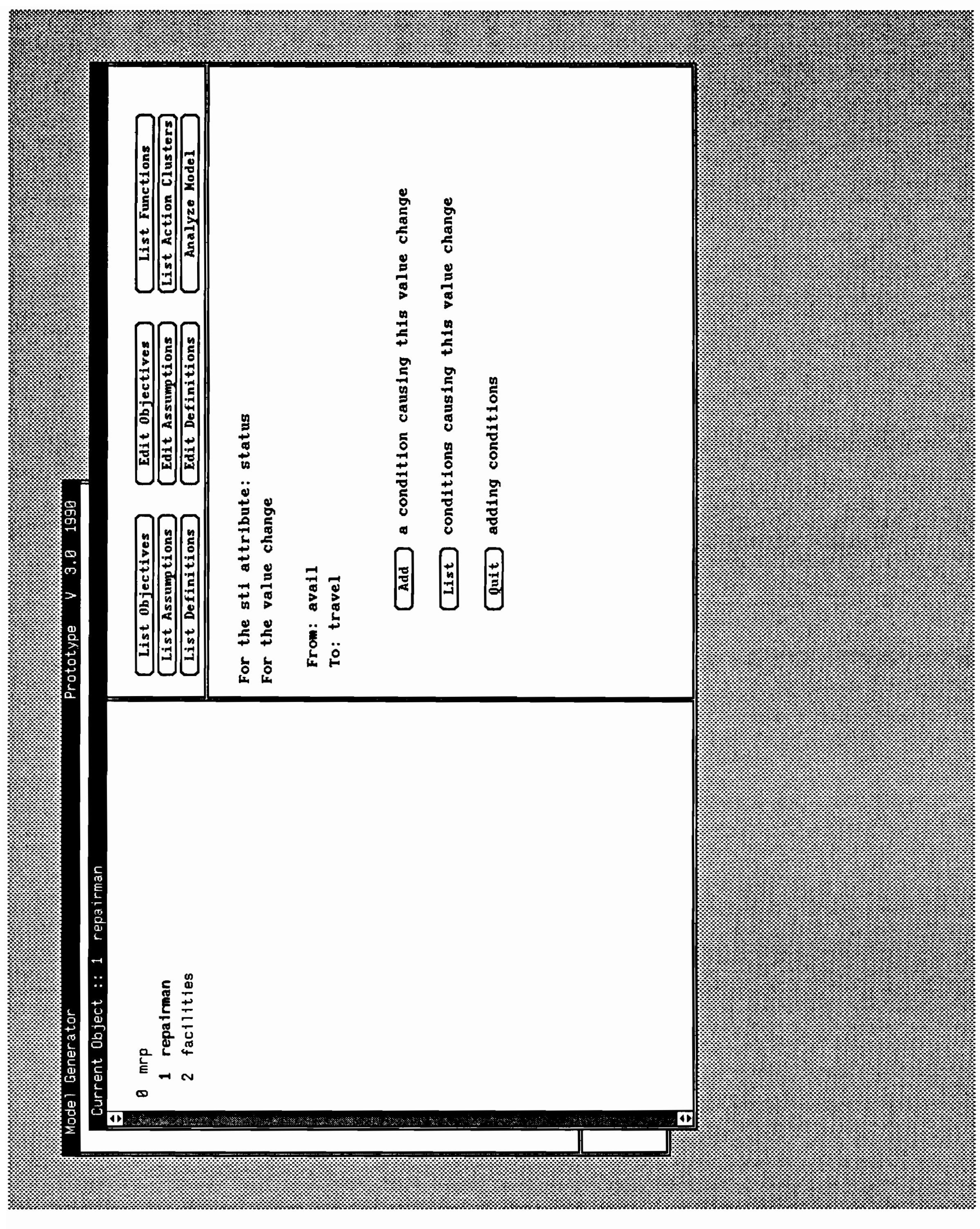

Figure 44. Specifying Conditions for Change in Value of a Status Transitional Indicative Attribute.

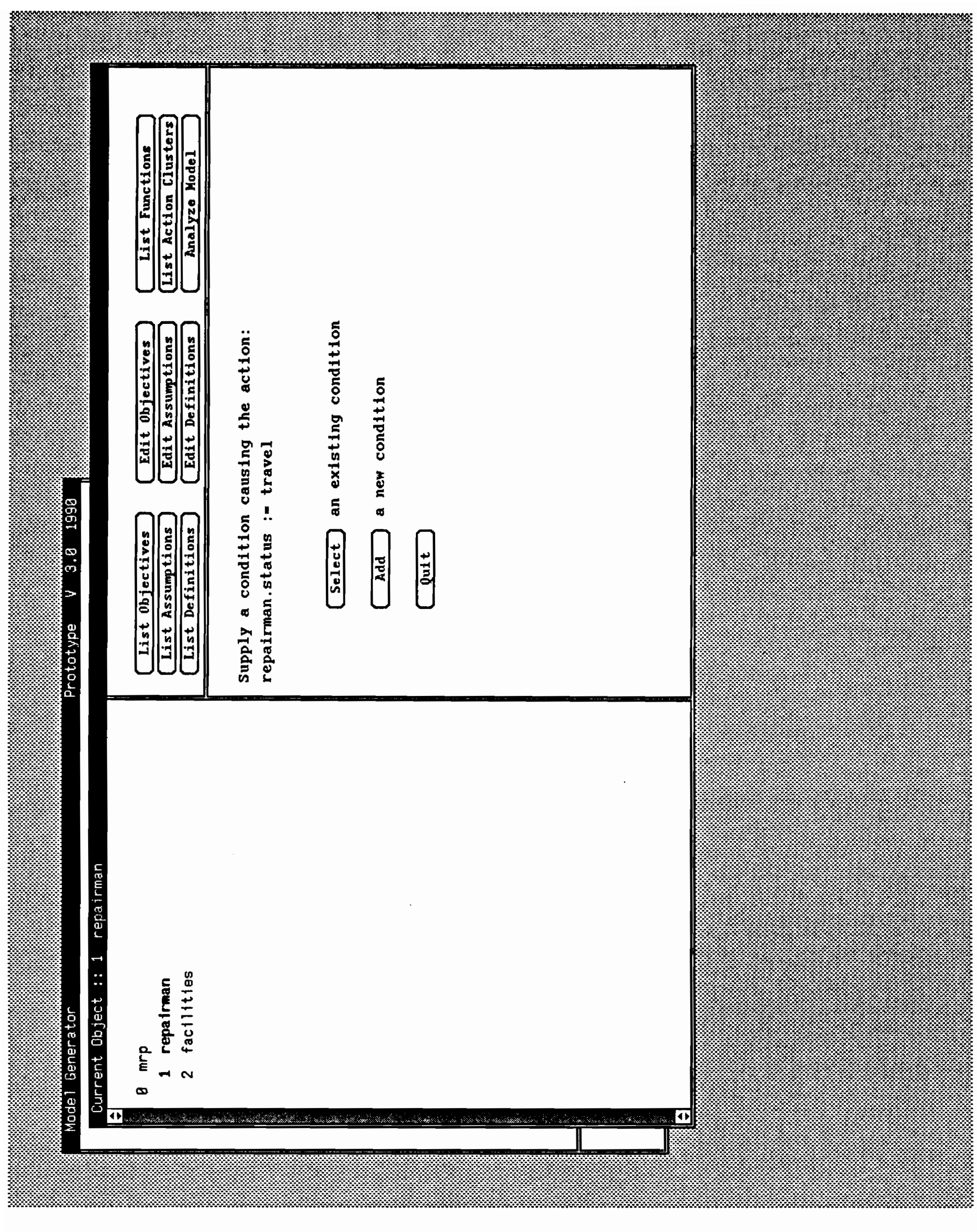

Figure 45. Supply Condition Menu.

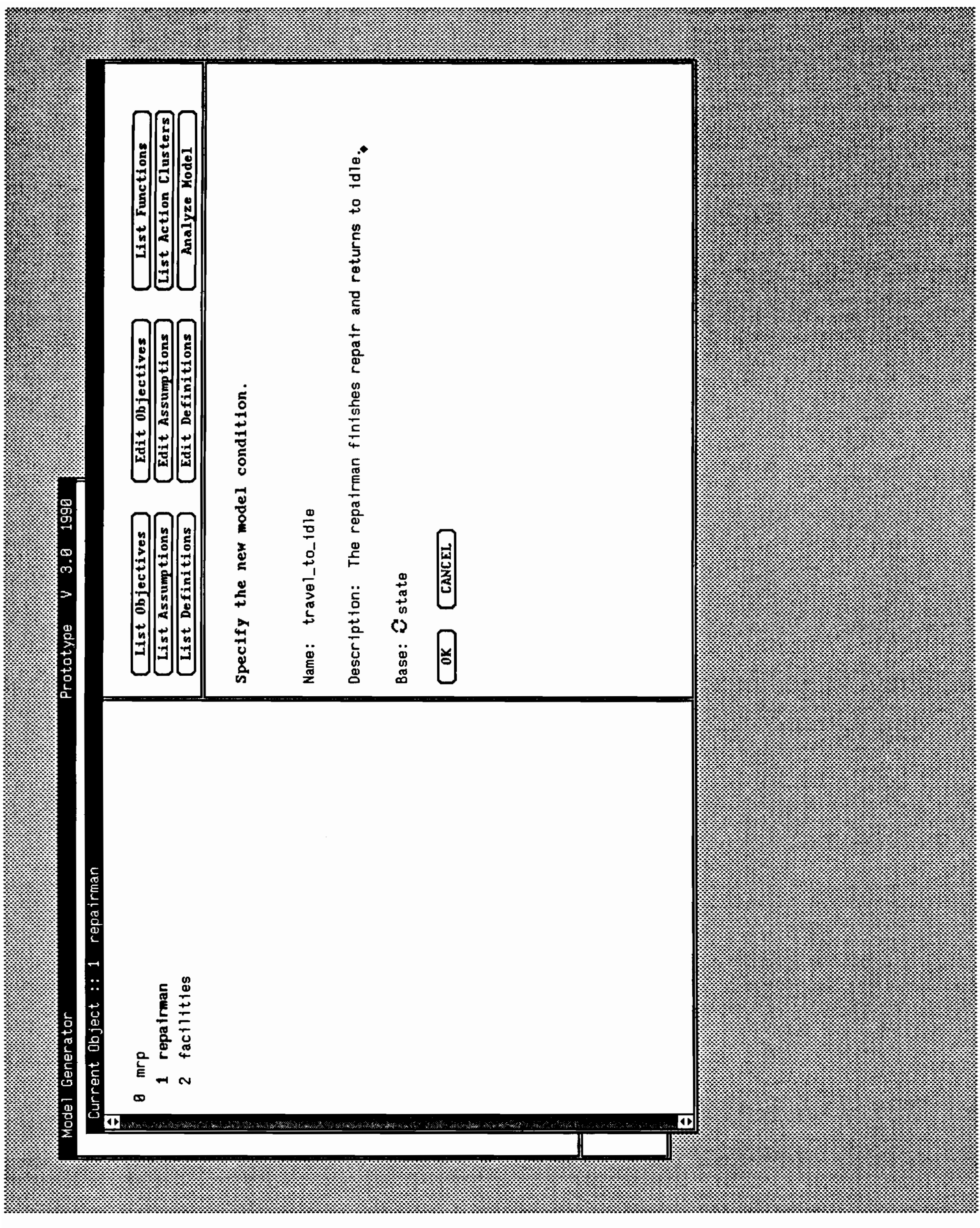

Figure 46. Specifying a New Condition.

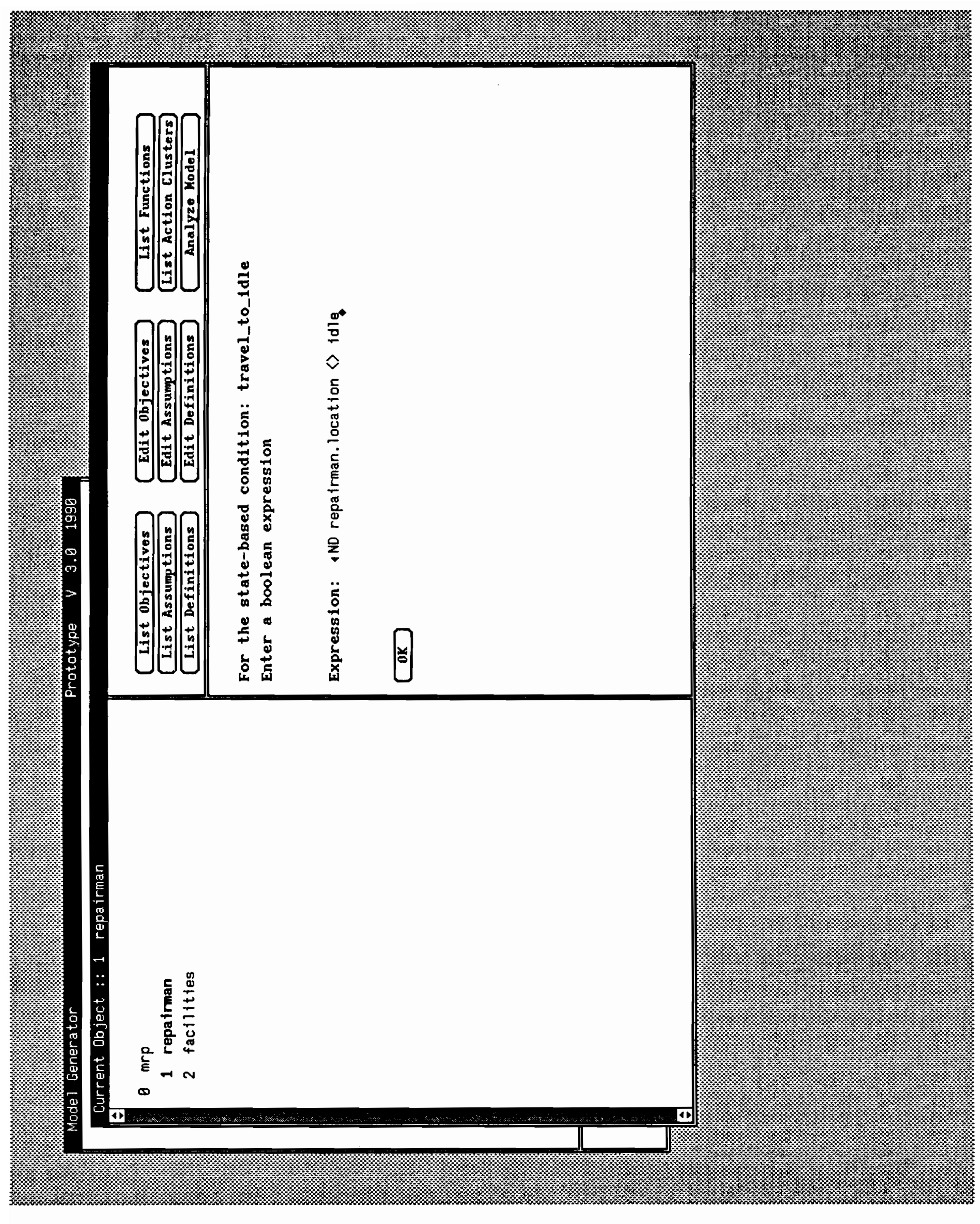

Figure 47. Entering a Boolean Expression for a State-based Condition.

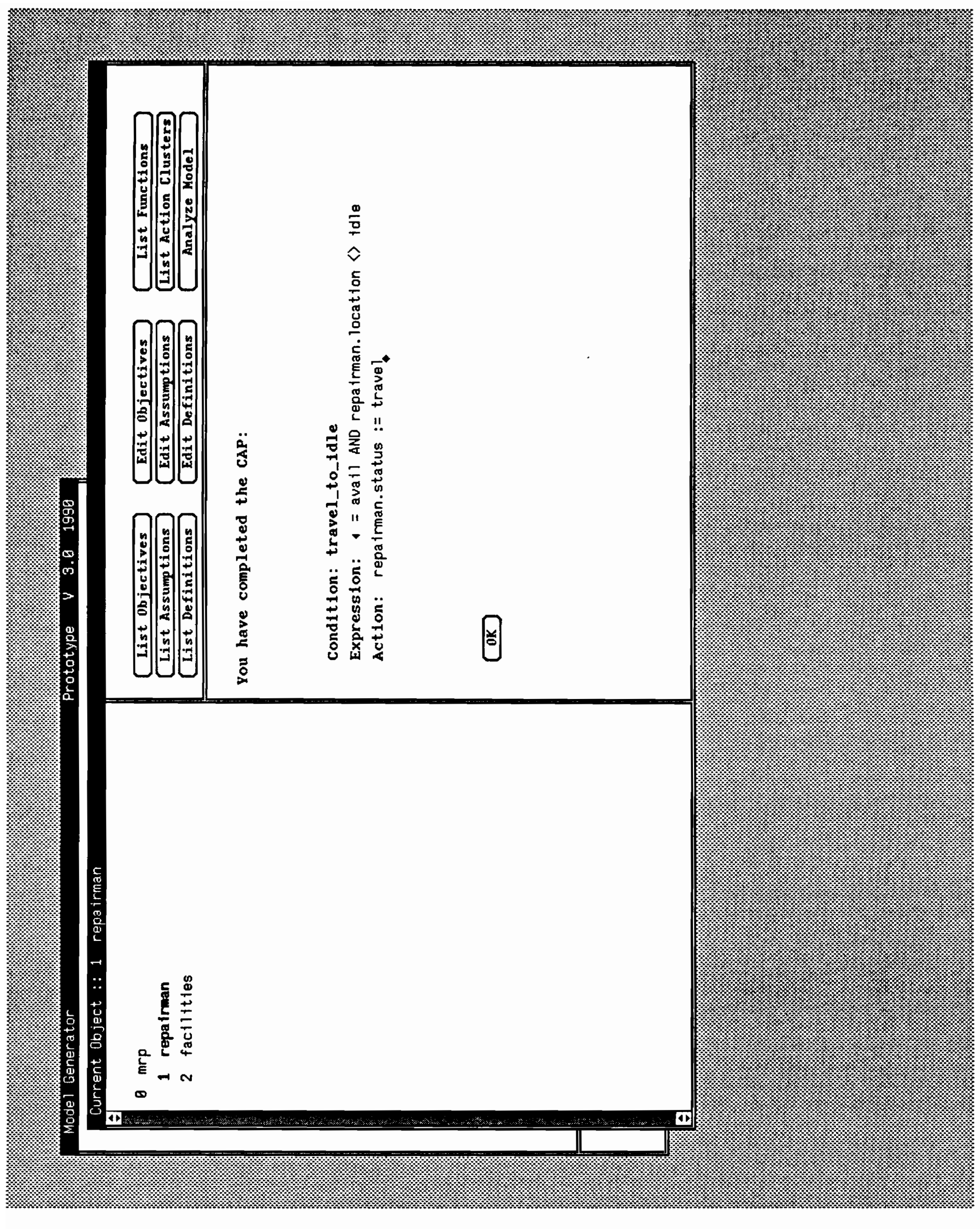

Figure 48. The Completed CAP Display.

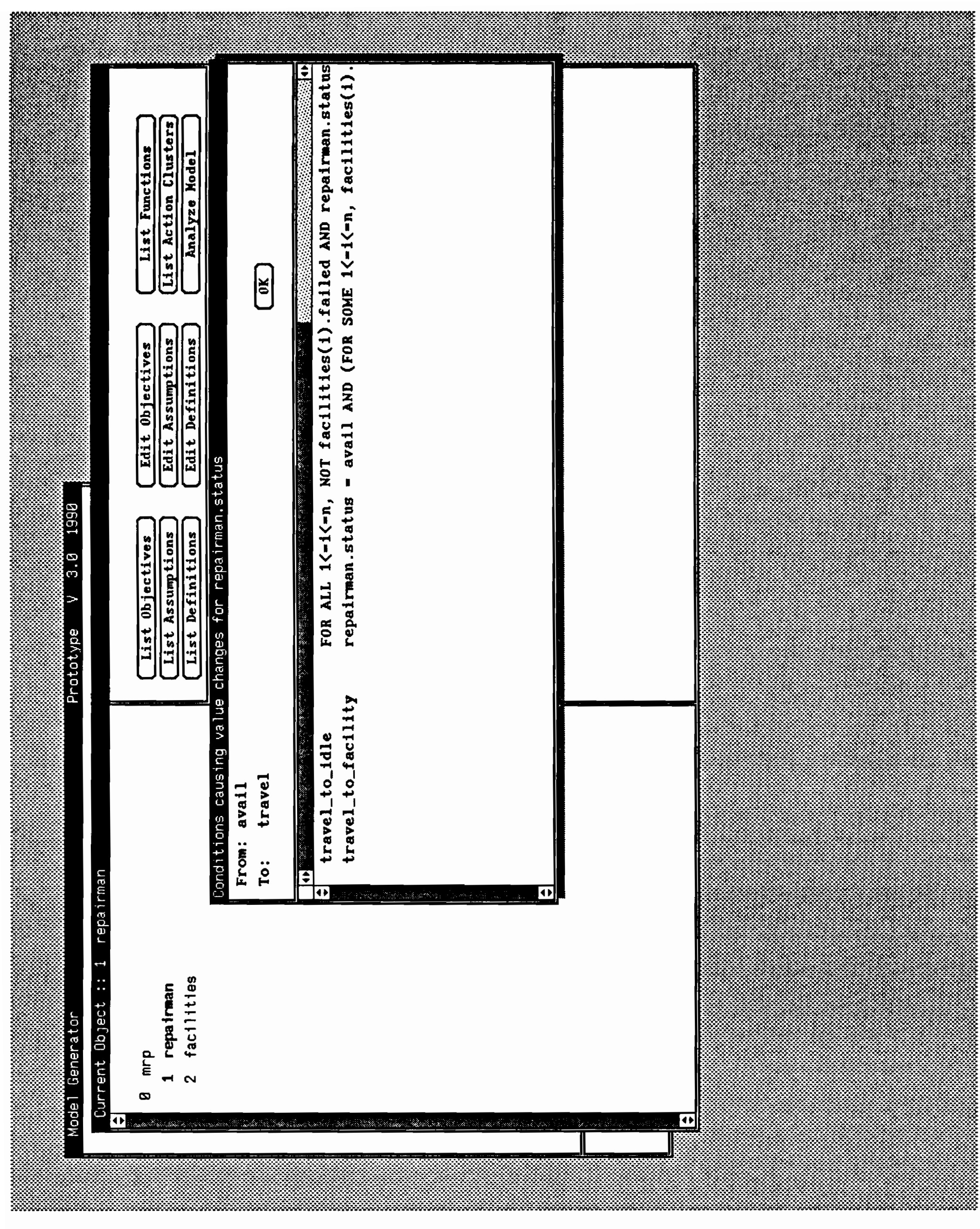

Figure 49. Listing Conditions Causing Value Change for a Status Transitional Indicative Attribute.

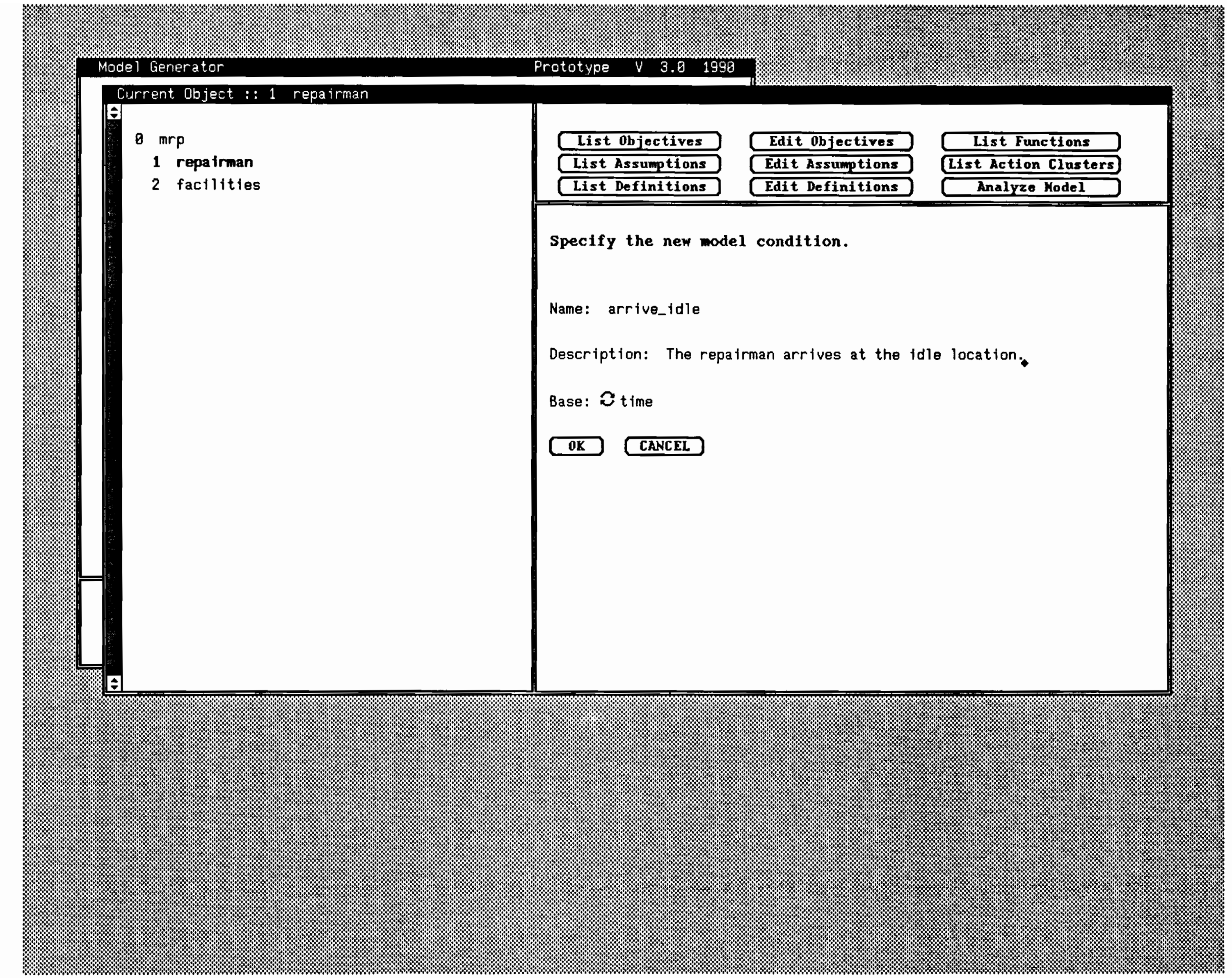

Figure 50. Specifying the Condition Arrive_idle.

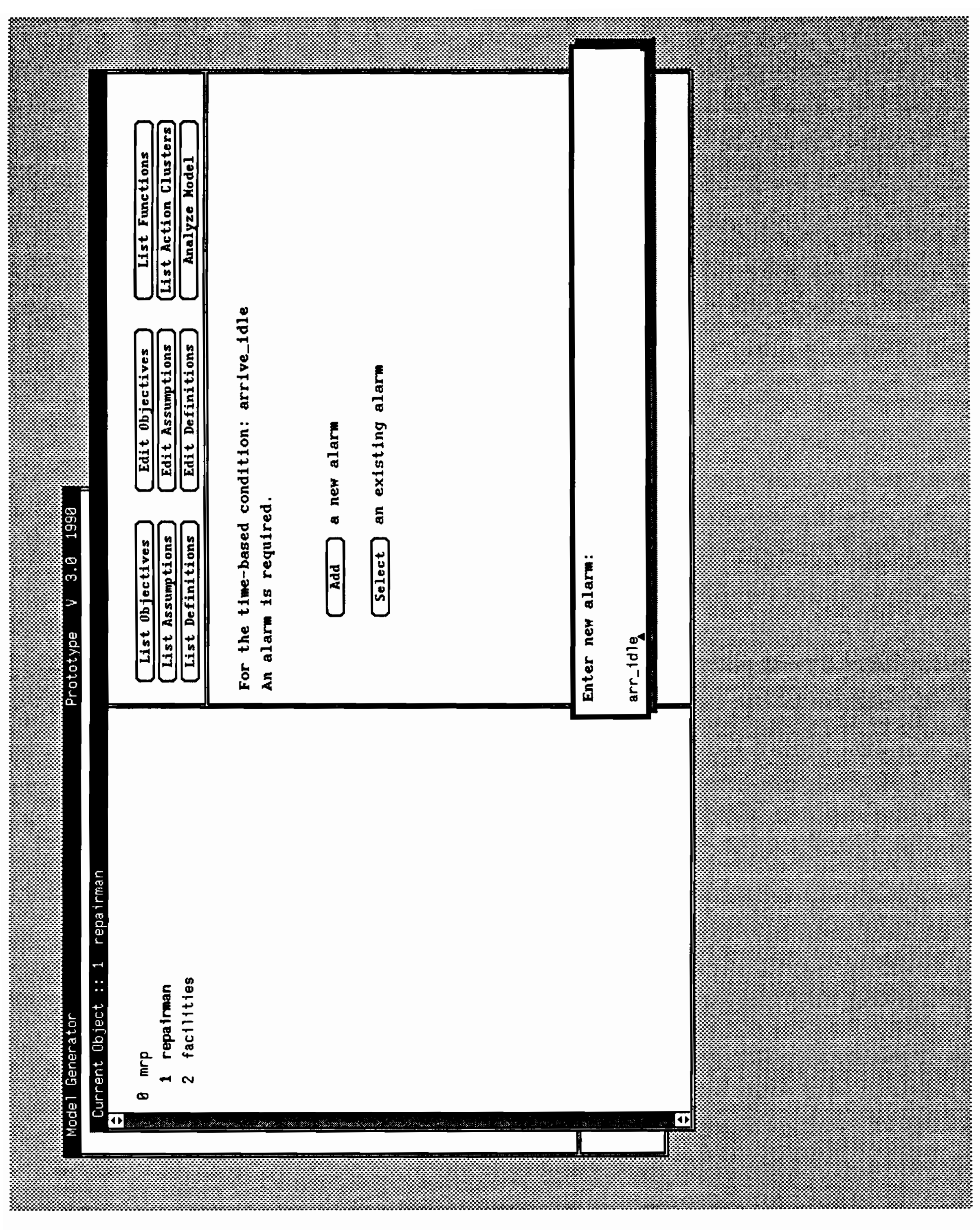

Figure 51. Specifying an Alarm for a Time-based Condition.

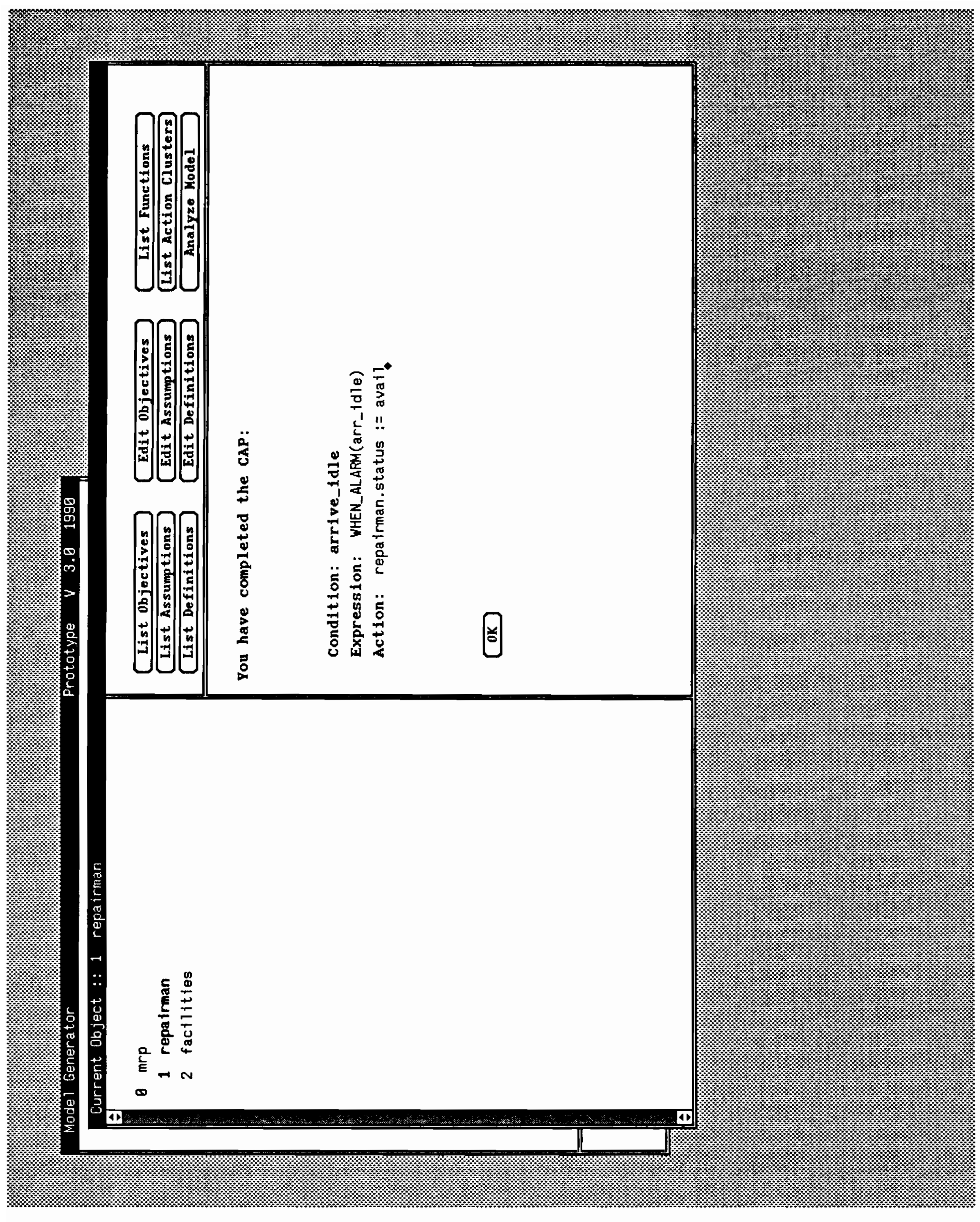

Figure 52. The Completed CAP Display for Arrive_idle.

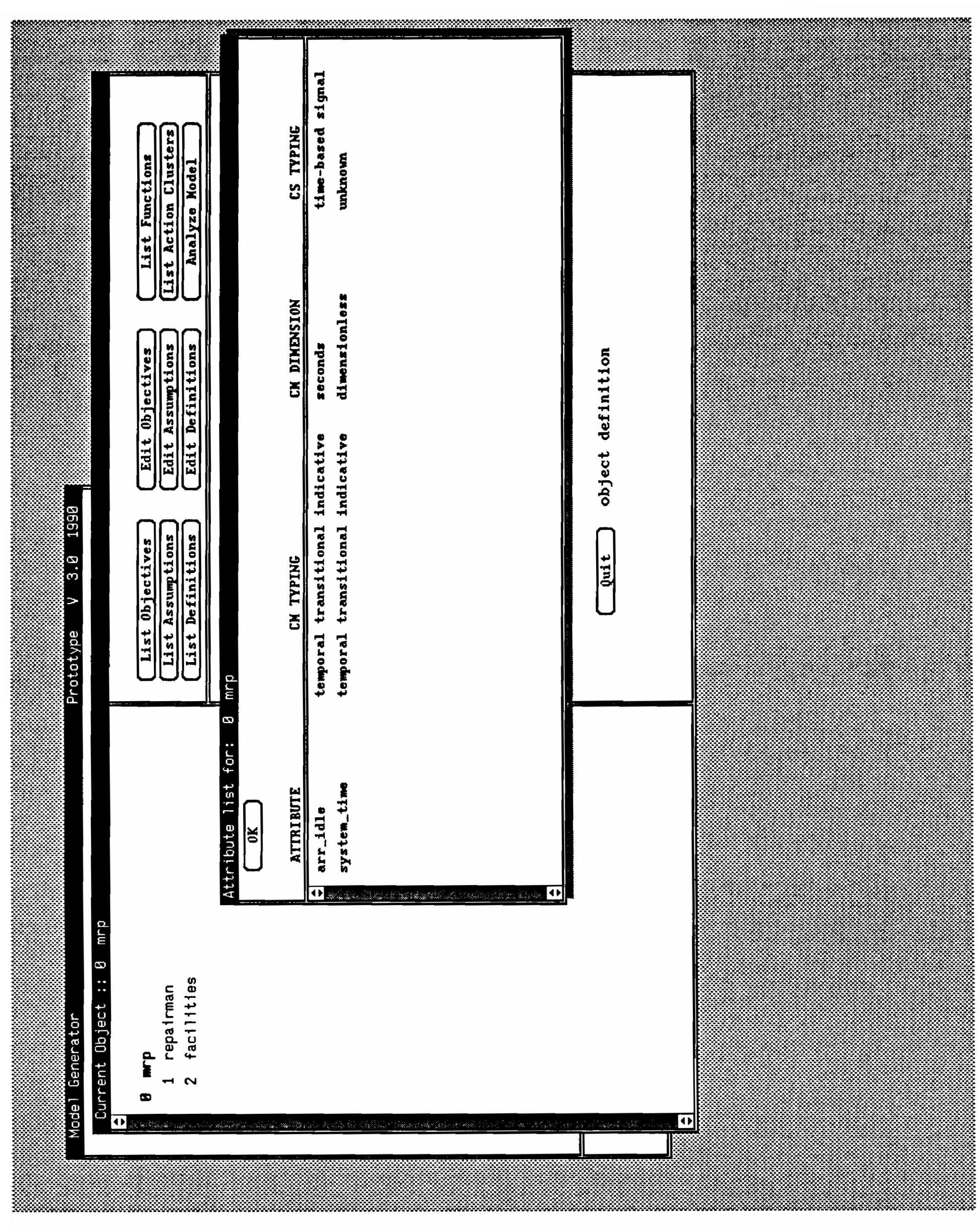

Figure 53. Alarms are Attributes of the Top Level Object.

6 Evaluation of the Model Generator

The final, and arguably most important, step in the design and implementation of a Model Generator for the SMDE is the evaluation of the prototype. Of course, only time and experience based on broad application of the prototype can provide the feedback necessary to assess this prototype effort. However, a preliminary evaluation of the prototype can, and should, be undertaken based on some set of well-defined criteria independent of the actual application of the prototype.

Since, from a specification perspective, the Model Generator is an extension of the Barger prototype, the criteria applied to the evaluation of the previous prototype should also be applied to the current one. Table 7 gives the criteria identified by Barger [1986] for a "good" specification and a "good" specification language. The concepts governing the nature and expressiveness of communication desired for a specification have not changed during the four years since this list was developed, therefore an evaluation of the specification derived in the Model Generator against these criteria is justified and given below.

Evaluation of the model specification:

• *Understandable* - The understandability of the Model Generator's specification is dependent on the level at which the specification is being examined. The text documentation provided by the Model Generator is simple and understandable by a variety of audiences, however, the understandability of the CAPs and action clusters is contingent upon an individual's familiarity with the Condition Specification.

Table 7. Specification Properties

[Barger 1986, p.14]

Properties of a "Good" Specification
• Understandable • Appropriate for many audiences • Presentable in varying levels of detail • Different views of same system can be presented • Separates implementation and description details • Includes description of environment • Information is localized • Easily modifiable • Analyzable
Properties of a "Good" Specification Language
• Encourages modularization • Encourages hierarchical descriptions • Allows use of terminology of current application • Mixture of formal and informal constructs • Encourages use of a development method • Produces documentation as a by-product • Easy to learn and use • Simple, precise, unambiguous syntax and semantics • Full range of acceptable system behavior can be described • Ability to describe variety of systems • Provides ability to access specification completeness • Nonprocedural
Properties of a "Good" Simulation Specification Language
• Independent of simulation programming languages • Allows expression of static and dynamic model properties • Facilitates model validation and verification • Nonprocedural

• *Appropriate for many audiences* - As stated above, the stratified (textual) documentation produced by the Model Generator is suitable for a variey of audiences such as upper-level and middle managers as well as simulation modelers. The CAPs and action clusters are more suited to simulation modelers than any other audience.

• *Presentable in various levels of detail* - The Model Generator utilizes the leveled dialogue technique established by Barger [1986]. As a modeler progresses through the dialogue (traversing the general tree which represents the model) specification details increase in complexity (i.e. description, naming, typing, conditions, and actions). The CM hierarchy allows a modeler to focus on any level of the development tree.

• *Presents different views of the same system* - The Model Generator provides an object oriented view of a system from a hierarchical decomposition perspective. However, the action clusters seem to provide a process oriented view of the system.

• *Separates implementation and description details* - The Condition Specification emphasizes the behavior of the system rather than the details of how the behavior is to be achieved. No provision for data structures or algorithms exists within the CS.

• *Includes a description of the system environment* - The CM provides environment description in the collection of model assumptions, definitions, and objectives. The CS provides an interface specification which indicates the nature of the communication between a model and its environment.

• *Information is localized* - The database is organized to localize model information according to objects, attributes, conditions, and actions. The CS localizes actions with conditions via the formation of action clusters.

• *Easily modifiable* - The Model Generator provides extensive capabilities for definition modification, but currently only provides specification changes by adding/deleting CAPs.

• *Analyzable* - Several works have shown the CS to provide a high degree of analyzability [Moose and Nance 1988; Nance and Overstreet 1984, 1985].

Evaluation of the dialogue:

• *Encourages modularization* - Specification information is modularized around model objects.

• *Encourages hierarchical description* - The CM explicitly advocates hierarchical decomposition of a model.

• *Allows use of terminology of current application* - Application terminology can be used in defining object names and attributes. However, the terminology of the CM and CS is evident in the Model Generator dialogue, and this may be disadvantageous.

• *Mixture of formal and informal constructs* - The Model Generator provides for informal entry of model documentation, but gradually forces some formality on the modeler when specifying model expressions that occur within conditions and actions.

• *Encourages use of a development method* - The dialogue forces the use of the CM as the model development method.

• *Produces documentation as a by-product* - Model documentation is a central point of focus for the Model Generator. A modeler is requested to provide descriptions of model objects, attributes,

conditions, and actions, thereby generating a textual description of the model. The CS of the model also provides a different form of model documentation.

- *Easy to learn and use* - Users of the Model Generator have found it easy to learn and use. However, these users are all members of the SMDE research group and are strongly familiar with the CM and the CS, therefore the sample may not be representative. Further experimentation among a variety of users is required to assess the Model Generator with respect to this criterion.

- *Simple, precise, unambiguous syntax and semantics* - Most responses require no knowledge of syntax, only "point and click" mouse movements. However, when entering expressions for model conditions and actions, a modeler must utilize some of the syntax dictated by the CS. The CS syntax has been evaluated as being very low level (see [Nance and Overstreet 1988; Derrick 1988]) and akin to that of a programming language. In light of this, the Model Generator attempts to handle much of the syntax and semantics of the CS automatically, requiring as little as possible from the modeler.

- *Full range of acceptable system behavior can be described* - Only extensive experimentation can determine if a CM-based Model Generator can provide this. To date, there is no evidence to the contrary.

- *Ability to describe variety of systems* - As above, only extensive experimentation can determine if a CM-based Model Generator can provide this. To date, there is no evidence to the contrary.

- *Provides ability to assess specification completeness* - As described in Section 4.3 the Model Generator analyzes specification completeness on two levels: (1) All specification information is present in the database, and (2) Attribute completeness (i.e. every object has an attribute, and every attribute is used in an expression).

- *Nonprocedural* - The dialogue contains a mixture of procedural and nonprocedural constructs and techniques. A model specification can be generated that is either highly procedural or highly nonprocedural. (The modeler's use of functions and sets seems to be a determining factor.)

- *Independent of simulation programming languages* - Both the CM and the CS are independent of simulation programming languages.

- *Allows expression of static and dynamic properties* - The model definition phase of the CM provides the description of static model properties. The model specification phase of the CM, utilizing the CS, provides the description of dynamic aspects of the model.

- *Facilitates model validation and verification* - The analyzability of the CS should facilitate model validation. However, the verification of a model as it is transformed from one representation to another in the SMDE is a subject of current research.

The Model Generator acquits itself well against these criteria. However, evaluation at this level is highly subjective. Only through extensive use can the final verdict be given as to the effectiveness with which the Model Generator performs its prescribed tasks.

Other than the machine repairman model, the Model Generator has been applied to the examples from [Overstreet 1982] and [Wallace 1985]. In all cases, the Model Generator is able to derive CS representations equivalent to those provided in the above works.

7 Summary and Conclusions

The primary goal of this research has been to develop a Model Generator that serves as a suitable platform for the assessment of the Condition Specification as the "target" language for simulation models developed under the framework of the Conical Methodology in the Simulation Model Development Environment. Further, the Model Generator should extend the concepts developed in the work of Hansen [1984] and Barger [1986], and utilize the capabilities for human-computer interaction and database design offered by the SUN environment.

A review of the literature reveals that the work in the area of simulation program generators offers little support by way of methodological guidance to a modeler. The current state of specification languages is such that the aid to a modeler provided by any one of these languages is little more than that afforded by a programming language. The future of specification portends augmenting formal approaches with graphical specifications and visual programming; however, these concepts are yet to be sufficiently developed to merit application in this research.

User interface researchers advocate the use of solid design principles in the development of user interfaces. Although little agreement exists regarding the precise definition of these design principles, the consensus among system designers seems to be that good user interfaces are sufficiently robust and flexible to accommodate a wide range of users. Procedures should be simple and easy

to learn while at the same time permitting more experienced users to take "short-cuts" through the system.

The new Model Generator prototype is an interactive tool which supports both the definition and specification phases of the Conical Methodology [Nance 1981a, 1984]. The model specification takes the form of a Condition Specification [Overstreet 1982]. The prototype appears to meet the requirements for a Model Generator in the SMDE.

7.1 Future Research

Directions of future research fall into two categories. Section 7.1.1 describes areas of improvement to the current prototype. Section 7.1.2 outlines a direction for research that builds on the information gathered by the Model Generator.

7.1.1 Prototype Improvements

Research directed toward improvements on the current Model Generator prototype should address the following areas:

(1) Improved Interface: The Model Generator interface is designed with *attention* to user interface principles, but certainly not with a strict adherence to any set of design principles. Production versions of all SMDE utilities should include interfaces designed by researchers well versed in Human Factors. Further, a production SMDE should have utility interfaces that are uniformly consistent.

(2) Improved Database: The Model Generator is developed as a prototype without any *a priori* knowledge of the information requirements of the tool. As a result, the database is developed incrementally with few overhauls. This results in some duplication of information in the database, as well as a mixture of normal forms within the relations. Future Model Generator research should address a redesign of the model database based on the information requirements identified by this research.

(3) Expression Parsing: Condition and action expressions need to be analyzed for syntactic and semantic correctness. Subsequent Model Generators should include an expression parser which could readily be developed using the UNIX facilities LEX and YACC. The original research goal included expression parsing as part of this prototype. However, the author's efforts in the current GPVSS research center on parsing a high level specification language using LEX and YACC. At this time, insufficient indications as to the future direction of the SMDE exist to merit duplication of the effort. The lack of an expression parser in the Model Generator does have some ramifications, particularly in the area of analysis of definition changes on the model specification. When a modeler enters an expression for a condition, the expression is stored in the database unparsed. The expression could contain references to one or more attributes. If a definition change later occurs to one of the attributes in the condition expression, there is no way for the Model Generator to specifically determine that the expression has been affected. Also, an expression may reference attributes which are undefined in the model. Parsing model expressions can add considerably to the power of the Model Generator in this area.

(4) Output: The Model Generator output in the SMDE should consist of two entities: a model specification in database form to be passed to the Model Analyzer, and a model specification document to be used as a communication vehicle for modelers and management. This research provides only for the former. Report generation techniques should be incorporated into future Model Generator prototypes.

(5) Unanswered Questions: Future Model Generator prototypes should also address the questions raised by this research, such as the utility of relational typing as it applies to model specification, the specification of function bodies, and the role of analysis within the Model Generator.

7.1.2 A New Direction

Simulations are some of the most CPU-intensive applications in use today. Many large simulations require days to execute. The rise of parallel processors offers tremendous potential for speeding up the execution of simulations. Much research is ongoing in the area of algorithms for parallel simulation; much less research is focusing on the assessment of a model's potential for parallelism. Given the high degree of analyzability demonstrated by the Condition Specification, the development of analysis techniques for simulation model specifications in a Condition Specification format with the goal of exploiting parallelism is certainly feasible. Metrics that indicate the level of inherent parallelism in a simulation model would be valuable to indicate to modelers and programmers the maximum speedup to expect when applying parallel processors to a simulation model. Other metrics based on a CS representation may provide an indication of a preferred architecture (shared memory vs. distributed memory), or a preferred algorithm (conservative vs. optimistic). This information could be utilized by the Model Translator in the SMDE to automatically generate a parallel simulation from a model developed using the Model Generator, thus achieving the automation-based paradigm and avoiding the pitfalls generally associated with parallel programming.

7.2 Conclusions

Although past SMDE research efforts have produced Model Generator prototypes, this research accomplishes two major tasks not achieved by other efforts:

(1) Both model definition and model specification are provided under the same umbrella, linking the Conical Methodology and Condition Specification. The precise nature of the relationship between the Conical Methodology and Condition Specification could not be completely evaluated prior to this work. To this end, the theoretical and philisophical avenues opened by this research go far beyond merely the development of a software tool.

(2) An experimental platform has been created to assess the capabilities of the CM/CS combination for the specification of models within the SMDE. For an environment designed to offer many forms of model specification, this prototype offers new insight into the capabilities and limitations of the CM and CS for a production SMDE.

Other contributions of this research include algorithms for the specification of:

- temporal transitional indicative attributes
- permanent indicative attributes
- alarms
- relational attributes
- time-based signals
- model input and output
- object creation and destruction
- model initialization and termination
- functions (headers)
- sets

This research also provides:

- definition of set manipulation primitives for the Condition Specification
- definition of data requirements for a Model Generator derived CS
- design of relational database according to data requirements
- initial evaluation of the utility of relational attributes in a CS
- initial evaluation of the role of analysis in the Model Generator
- exploitation of SUN technology to provide multitasking in a window-based prototype

Evaluation of this prototype, coupled with the ongoing research and extensions of GPVSS [Bishop 1989], should furnish valuable guidance for future prototype development.

Bibliography

Alexander, C. (1964). *Notes on the Synthesis of Form,* Harvard University Press, Cambridge, MA.

Alford, M. W. (1977). "A Requirement Engineering Methodology for Realtime Processing Environments," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, pp. 60-69, January.

Alford, M. W. (1985). "SREM at the Age of Eight: The Distributed Computing Design System," *IEEE Computer,* Vol. 18, No. 4, pp. 36-46, April.

Badre, A. N., Shneiderman, B. (1982). *Directions in Human/Computer Interaction,* Ablex Publishing Corporation, Norwood, NJ.

Balci, O. (1986a). "Requirements for Model Development Environments," *Computers and Operations Research,* Vol. 13, No. 1, pp. 53-67.

Balci, O. (1986b). "Guidelines for Successful Simulation Studies," Technical Report TR-85-2, Department of Computer Science, Virginia Tech, Blacksburg, VA, September.

Balci, O., and Nance, R. E. (1987a). "Simulation Support: Prototyping the Automation-Based Paradigm," In: *Proceedings of the 1987 Winter Simulation Conference,* pp. 495-501.

Balci, O., and Nance, R. E. (1987b). "Simulation Model Development Environments: A Research Prototype," *Journal of the Operational Research Society,* Vol. 38, No. 8, pp. 753-763.

Balmer, D.W., and Paul, R.J. (1986). "CASM - The Right Environment for Simulation," *Journal of the Operational Research Society,* Vol. 37, No. 5, pp. 443-452.

Balzer, R., and Goldman, N. (1979). "Principles of Good Software Specification and Their Implications for Specification Languages," In: *Proceedings of the IEEE Conference on Specification for Reliable Software,* pp. 58-67, April.

Balzer, R. (1980). "An Implementation Methodology for Semantic Database Models," *Entity-Relationship Approach to Systems Analysis and Design,* P. P. Chen ed., North-Holland Publishing Company, Amsterdam, pp. 433-444.

Balzer, R., Wile, D. S., and Goldman, N. (1982). "Operational Specification as the Basis for Rapid Prototyping," *ACM SIGSOFT Software Engineering Notes,* Vol. 7, No. 5, pp. 3-16, December.

Balzer, R., Cheatham, T.E., and Green, C. (1983). "Software Technology in the 1990's: Using a New Paradigm," *Computer,* Vol. 16, No. 11, pp. 39-45, November.

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

Bell, T. E., Bixler, D. C., and Dyer, M. E. (1977). "An Extendable Approach to Computer-aided Software Requirements Engineering," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, pp. 49-60, January.

Berzins, V. (1985). "Analysis and Design in MSG.84," In: *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 8, pp. 657-670, August.

Birtwistle, G., Joyce, J., and Wyvill, B.L.M., (1984). "ANDES: An Environment for Animated Descrete Event Simulation," United Kingdom Simulation Conference, Bath, September.

Bishop, J.L. (1989). "General Purpose Visual Simulation System," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, June.

Borgida, A., Greenspan, S., and Mylopoulos, J. (1985). "Knowledge Representation as the Basis for Requirements Specification," *IEEE Computer,* Vol. 18, No. 4, pp. 82-91, April.

Brackett, J.W. (1988). "Formal Specification Languages: A Marketplace Failure; A Position Paper," In: *Proceedings of the 1988 IEEE International Conference on Computer Languages,* Miami, FL, p. 161, October 9-13.

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

Brown, G.P., Carling, R.T., Herot, C.F., Kramlich, D.A., and Souza, P. (1985). "Program Visualization: Graphical Support for Software Development," *IEEE Computer,* Vol. 18, No. 8, pp. 27-35, August.

Chen, P.P. (1976). "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems,* Vol. 1, No. 1, pp. 9-36, March.

Clementson, A.T. (1973). "Extended Control and Simulation Language," University of Birmingham, Birmingham England.

Date, C. J. (1986). *An Introduction to Database Systems,* Addison-Wesley Publishing Company, Reading, MA, Volume 1, Fourth Edition.

Davis, C. G., and Vick, C. R. (1977). "The Software Development System," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, pp. 69-84, January.

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

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

Faught, W. S., Klahr, P., and Martins, G. R. (1980). "An Artificial Intelligence Approach to Large Scale Simulation," In: *Proceedings of the Summer Simulation Conference,* Seattle, Washington, pp. 231-235, August 25-27.

Feather, M. S. (1983). "Reuse in the Context of a Transformation Based Methodology," In: *Proceedings of the Workshop on Reusability in Programming,* Newport, Rhode Island, pp. 50-58, September 7-9.

Fishman, G.S. (1983). *Concepts and Methods in Discrete Event Digital Simulation,* John Wiley and Sons, New York.

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

Freeman, P. (1983). "Fundamentals of Design," In: *Tutorial on Software Design,* Peter Freeman and Anthony Wasserman, eds., IEEE Computer Society Press, pp. 2-22.

Gaines, B. R., and Facey, P. V. (1975). "Some Experience in Interactive System development and Application," In: *Proceedings of the IEEE,* 63, 6, pp. 894-911, June.

Garrison, W.J. (1989). "NETWORK II.5 Tutorial: Network Modeling - Without Programming," In: *Proceedings of the 1989 Winter Simulation Conference,* Washington, DC, pp. 205-214, December 4-6.

Gehani, Narain (1982). "Specifications: Formal and Informal -- A Case Study," In: *Software - Practice and Experience,* Vol. 12, pp. 433-444.

Gillet, W.D., and Kimura, T.D. (1986). "Parsing Two-Dimensional Languages," *IEEE COMPSAC,* Chicago, IL, pp. 472-477, November.

Goldman, N., and Wile, D. (1980). "A Relational Database Foundation for Process Specification," *Entity-Relationship Approach to Systems Analysis and Design,* P. P. Chen ed., North-Holland Publishing Company, Amsterdam, pp. 413-432.

Hamilton, M. and Zeldin, S. (1976). "Higher Order Software - A Methodology for Defining Software," *IEEE Transactions on Software Engineering,* Vol. SE-2, No. 1, pp. 173-190, January.

Hamilton, M., and Zeldin, S. (1983). "The Relationship Between Design and Verification," In: *Tutorial on Software Design,* Peter Freeman and Anthony Wasserman, eds., IEEE Computer Society Press, pp. 641-668.

Hansen, R. H. (1984). "The Model Generator: A Crucial Element of the Model Development Environment," Technical Report CS84008-R, Department of Computer Science, Virginia Tech, Blacksburg, VA, August.

Hansen, W. J. (1971). "User Engineering Principles for Interactive Systems," In: *Proceedings of the Fall Joint Computer Conference,* 39, AFIPS Press, pp. 523-532, Montvale, NJ.

Harel D. (1988). "On Visual Formalisms," *Communications of the ACM,* Vol. 31, No. 5, pp. 514-530, May.

Harel D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., and Shtul-Trauring, A. (1988). "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," In: *Proceedings of the 10th International Conference on Software Engineering,* Singapore, pp. April 13-15.

Heacox, H. C. (1979). "RDL - A Language for Software Development," *ACM SIGPLAN Notices,* Vol. 14, pp. 71-79, December.

Hills, P.R., and Poole, T.G. (1969). "A Method for Simplifying the Production of Computer Simulation Models," In: *Proceedings of the TIMS Tenth America Meeting,* Atlanta, GA, October 1-3.

Hix, D., and Siochi, A. (1989). "Representation of Direct Manipulation Human-Computer Interfaces: An Introduction," Unpublished manuscript. Viginia Polytechnic Institute and State University, Blacksburg, VA, March.

Holbaek-Hanssen, E., Handlykken, P. and Nygaars, K. (1977). *System Description and the Delta Language, Report No. 4,* Robin Hills (Consultants) Ltd., Surrey, England.

Ichikawa, T., and Hirakawa, M. (1987). "Visual Programming - Toward Realization of User-Friendly Programming Environment," *Proceedings of FJCC,* pp. 129-137. October.

Inoue, K., Ogihara, T., Kikuno, T., and Torili, K. (1989). "A Formal Adaptation Method for Process Descriptions," In: *Proceedings of the 11th International Conference on Software Engineering,* pp. 145-153, May 16-18.

Jadoul, L., Duponcheel, L., and Puymbroeck, W.V. (1989). "An Algebraic Data Type Specification Language and its Rapid Prototyping Environment," In: *Proceedings of the 11th International Conference on Software Engineering,* pp. 74-84, May 16-18.

Katayama, T. (1989). "A Hierarchical and Functional Software Process Description and its Enaction," In: *Proceedings of the 11th International Conference on Software Engineering,* pp. 343-352, May 16-18.

Lenz, J.E. (1989). "The MAST Simulation Environment," *Proceedings of the 1989 Winter Simulation Conference,* Washington, DC, pp. 243-248, December 4-6.

Liskov, B. H., and Zilles, S. N. (1975). "Specification Techniques for Data Abstraction," *IEEE Transactions on Software Engineering,* Vol. SE-1, No. 1, pp. 7-19, March.

Lodding, K.N. (1982). "Icons: A Visual Man-Machine Interface," In: *Proceedings of the 1982 National Computer Graphics Association,* Fairfax, VA, pp. 221-233.

London, P. E., and Feather, M. S. (1982). "Implementing Specification Freedoms," *Science of Computer Programming,* Vol. 2, North-Holland Publishing Company, Amsterdam, pp. 91-131.

Martin, J. (1982). *Application Development Without Programmers,* Prentice-Hall, Inc., Englewood Cliffs, NJ.

Martin, J., and McClure C. (1985a). *Diagramming Techniques for Analysts and Programmers,* Prentice-Hall, Inc., Englewood Cliffs, NJ.

Martin, J. (1985b). *System Design From Provably Correct Constructs,* Prentice-Hall, Inc., Englewood Cliffs, NJ.

Mathewson, S.C. (1974). "Simulation Program Generators," *Simulation,* Vol. 23, No. 6, pp. 181-189.

Mathewson, S.C. (1975). "Program Generators," In: *Proceedings of the European Computing Conference on Interactive Systems,* Brunel University, pp. 423-439.

Mathewson, S.C. (1984). "Simulation Program Generators and Animation on a PC," In: *OR Society Conference on 16 Bit Microcomputers,* November.

Miller G. A. (1956). "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review,* Vol. 63, pp. 81-97.

Molich, R., and Nielsen, J. (1990). "Improving a Human-Computer Dialogue," *Communications of the ACM,* Vol. 33, No. 3, pp. 338-348, March.

Moose, R.L., and Nance, R.E. (1988). "The Design and Development of an Analyzer for Discrete Event Model Specifications," Technical Report SRC-87-010, Systems Research Center and Department of Computer Science, Virginia Tech, Blacksburg, VA, June.

Nance, R. E. (1971). "On Time Flow Mechanisms for Discrete Event Simulations," *Management Science,* Vol. 18, No. 1, pp. 59-93, September.

Nance, R. E. (1977). "The Feasibility of and Methodology for Developing Federal Documentation Standards for Simulation Models," Final Report to the National Bureau of Standards, Department of Computer Science, Virginia Tech, Blacksburg, VA, June.

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, March.

Nance, R. E. (1981b). "The Time and State Relationships in Simulation Modeling," *Communications of the ACM,* Vol. 24, No. 4, pp. 173-179, April.

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

Nance, R. E. (1984). "Model Development Revisited," In: *Proceedings of the 1984 Winter Simulation Conference,* Dallas, TX, pp. 75-80, November 28-30.

Nance, R. E., Balci O., and Moose, R. L. (1984). "Evaluation of the UNIX Host for a Model Development Environment," In: *Proceedings of the 1984 Winter Simulation Conference,* Dallas, TX, pp. 577-584, November 28-30.

Nance, R. E., and Overstreet C. M. (1984). "Graph-Based Diagnosis of Discrete Event Model Specification," Technical Report SRC-85-003, Systems Research Center and Department of Computer Science, Virginia Tech, Blacksburg, VA, June.

Nance, R. E. (1987). "The Conical Methodology: A Framework for Simulation Model Development," In: *Proceedings of the Conference on Methodology and Validation,* (ESC, Orlando, FL, April 6-9). Published as *Simulation Series 19,* 1, pp. 38-43, SCS, San Diego, CA, January.

Nance, R. E., and Overstreet C. M. (1988). "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," *Transactions of the Society for Computer Simulation,* The Society for Computer Simulation, Vol. 4, No. 1, pp. 33-57.

Nance, R. E. (1989). "Contemplations of a Simulation Navel or Recognizing the Seers Among the Peers," *Simulation Digest,* SIGSIM Vol. 20, No. 3, TCSIM Vol. 31, pp. 53-59, Fall.

O'brien, P. D. (1983). "An Integrated Interactive Design Environment for TAXIS," *SOFTFAIR,* Arlington, VA, pp. 298-306, July 25-28.

Oldfather, P.M., Ginsberg, P.L., and Markowitz, H.M. (1966). "Programming by Questionnaire: How to Construct a Program Generator," RAND Report RM-5129-PR, November.

Oldfather, P.M., Ginsberg, A.S., Love, P.L., and Markowitz, H.M. (1967). "Programming by Questionnaire: The Job Shop Simulation Program Generator," RAND Report RM-5162-PR, July.

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

Overstreet, C. M., and Nance, R. E. (1985). "A Specification Language to Assist in Analysis of Discrete Event Simulation Models," *Communications of the ACM,* Vol. 28, No. 2, pp. 190-201, February.

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

Palm, D. C. (1947). "The Distribution of Repairmen in Servicing Automatic Machines," *Industritidningen Norden 175,* 75, (in Sweedish).

Paul, R.J., and Chew, S.T. (1987). "Simulation Modelling Using an Interactive Simulation Program Generator," *Journal of the Operational Research Society,* Vol. 38, No. 8, pp. 735-752.

Reddy, Y.V., Fox, M.S., Husain, N., and McRoberts, M. (1986). "The Knowledge-Based Simulation System," *IEEE Software,* Vol. 3, pp. 26-37.

Reiss, S.P. (1985). "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering,* SE-11, No. 3, pp. 276-285, March.

Reiss, S.P. (1986). "GARDEN Tools: Support for Graphical Programming," In: *Proceedings of the IFIP International Workshop on Advanced Programming Environments,* pp. 59-72, June.

Riddle, W. E., et al (1978a). "A Descriptive Scheme to Aid the Design of Collections of Concurrent Processes," In: *Proceedings AFIPS National Computer Conference,* Anaheim, CA, pp. 549-554, June.

Riddle, W. E., et al (1978b). "Behavior Modeling During Software Design," *IEEE Transactions on Software Engineering,* Vol. SE-4, No. 4, pp. 283-292, July.

Riddle, W. E. (1979). "An Event-Based Design Methodology Supported by DREAM," In: *Tutorial on Software design,* Peter Freeman and Anthony Wasserman, eds., IEEE Computer Society Press, pp. 378-392.

Riddle, W. E. (1980). "An Assessment of DREAM," *Software Engineering Environments,* Horst HUNKE, ed., North-Holland Publishing Company, Amsterdam, pp. 191-221.

Rohrbough, M.C. (1989). "Introduction to SIMFACTORY II.5," In: *Proceedings of the 1989 Winter Simulation Conference,* Washington, DC, pp. 201-204, December 4-6.

Roth, P.F., Gass, S.I., and Lemoine, A.J. (1978). "Some Considerations for Improving Federal Modeling," In: *Proceedings of the 1978 Winter Simulation Conference,* Miami Beach, Florida, December.

Shannon, R.E. (1975). *Systems Simulation, The Art and Science,* Prentice-Hall, Englewood Cliffs, N.J.

Scheffer, P. A., Stone, A. H., Rzepka, W. E. (1985). "A Case Study of SREM," *IEEE Computer,* Vol. 18, No. 4, pp. 47-54, April.

Shneiderman B. (1980). *Software Psychology - Human Factors in Computer and Information Systems,* Winthrop Publishers, Inc., Cambridge, MA.

Shneiderman B. (1987). *Designing the User Interface: Strategies for Effective Human-Computer Interaction,* Addison-Wesley Publishing, Reading, MA.

Silverberg, B. A. (1981). "An Overview of the SRI Hierarchical Development Methodology," *Software Engineering Environments,* Horst HUNKE, ed., North-Holland Publishing Company, Amsterdam, pp. 235-252.

Standridge, C.R., Vaughn, D.K., and Sale, M.L., (1985). "A Tutorial on TESS: The Extended Simulation System," In: *Proceedings of the 1985 Winter Simulation Conference,* San Francisco, CA, pp. 73-79, December.

Stoegerer, J. K. (1984). "A Comprehensive Approach to Specification Languages," *Australian Computer Journal,* Vol. 16, No. 1, pp. 1-13, February.

Sun Microsystems (1988a). *SunView 1 Programmer's Guide,* Rev A, Sun Microsystems, Inc., Mountain View, CA, May 9.

Sun Microsystems (1988b). *SunView 1 Beginner's Guide,* Rev A, Sun Microsystems, Inc., Mountain View, CA, May 9.

Thompson, M.B. (1989). "AutoMod II: The System Builder," *Proceedings of the 1989 Winter Simulation Conference,* Washington, DC, pp. 235-242, December 4-6.

Tocher, K.D.T. (1976). "Some Techniques of Model Building," *Proceedings IBM Scientific Computing Symposium on Simulation Models and Planning,* White Plains, NY, pp. 119-155, December 7-9.

Teichrow, D., and Hershey, E. A. III (1977). "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, pp. 41-48, January.

Teichrow, D., et al. (1980). "Application of the Entity-Relationship Approach to Information Processing Systems Modeling," In: *Entity-Relationship Approach to Systems Analysis and Design,* P. P. Chen, ed., North-Holland Publishing Company, Amsterdam, pp. 15-38.

Taylor, T., and Standish, T. A. (1982). "Initial Thoughts on Rapid Prototyping Techniques," *ACM SIGSOFT Software Engineering Notes,* Vol. 7, pp. 160-166.

Unger, B., Dewar, A., Cleary, J., and Birtwistle, G. (1986). "A Distributed Software Prototyping and Simulation Environment: JADE," In: *Proceedings of the Conference on Intellegent Simulation Environments,* pp. 63-71, January.

Vidallon, C. (1980). "GASSNOL: A Computer Subsystem for the Generation of Network Oriented Languages with Syntax and Semantic Analysis," *Simulation '80,* Reprint, Interlaken, Switzerland, June 25-27.

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

Wasserman, A. I. (1973). "The Design of Idiot-Proof Interactive Systems," In: *Proceedings of the National Computer Conference,* 42, AFIPS Press, Montvale, NJ.

Wasserman, A. I. (1983). "Information System Design Methodology," In: *Tutorial on Software design,* Peter Freeman and Anthony Wasserman, eds., IEEE Computer Society Press, pp. 43-62.

Winters, E. W. (1979). "An Analysis of the Capabilities of PSL: A Language for System Requirements and Specifications, " *IEEE COMPSAC,* Chicago, IL, pp. 283-288, November.

Yau, S.S., and Jia, X. (1988). "Visual Languages and Software Specifications," In: *Proceedings of the 1988 IEEE International Conference on Computer Languages,* Miami, FL, pp. 322-328, October 9-13.

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

Yestingsmeir, J. (1984). "Human Factors Considerations in Development of Interactive Software," *SIGCHI Bulletin,* Vol. 16, No. 1, pp. 24-27, July.

Zave, P. (1979). "A Comprehensive Approach to Requirements Problems," *IEEE COMPSAC,* Chicago, Illinois, pp. 117-122, November.

Zave, P. (1982). "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Transactions on Software Engineering,* Vol. SE-8, No. 3, pp. 250-269, May.

Zave, P. (1984a). "The Operational Versus the Conventional Approach to Software Development," *CACM,* Vol. 27, No. 2, pp. 104-118, February.

Zave, P. (1984b). "An Overview of the PAISLey Project - 1984," *ACM SIGSOFT Software Engineering Notes,* Vol. 9, No. 4, pp. 12-19, July.

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

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

TASK: Select_Object		
User Actions	**Feedback**	**System State**
~[object_name]; $M_L v \wedge$	object_name ! object_name appears in frame	Object becomes current (selected) object

Figure 54. UAN : Select_Object

TASK: Attach_Attributes		
User Actions	**Feedback**	**System State**
~[attach_button]; M_Lv∧	attach_button ! -! Remove(def_menu) Display(att_menu)	Attach attribute enabled
K(name); Rv∧	name is displayed	
K(description);	description is displayed	
~[type_selector]; M_Lv∧*	type′ is displayed	
~[dim_selector]; M_Lv∧*	dimension′ is displayed	
~[ok_button]; M_Lv∧	ok_button ! -! Remove(att_menu) Display(def_menu)	Attribute is attached to current object

Figure 55. UAN : Attach_Attributes

TASK: Make_Subobject		
User Actions	**Feedback**	**System State**
~[make_button]; M_Lv∧	make_button ! -! Display(name_menu)	Make subobject enabled
~[name_menu]; K(name); Rv∧	Remove(name_menu) Object appears in heirarchy and frame	new object is current object

Figure 56. UAN : Make_Subobject

TASK: List_Attributes		
User Actions	**Feedback**	**System State**
~[list_button]; M_Lv∧	list_button ! -! Display(att_list)	List active
~[ok_button]; M_Lv∧	Remove(att_list)	List inactive

Figure 57. UAN : List_Attributes

TASK: Get_Condition		
User Actions	**Feedback**	**System State**
Select_Existing_Condition | Select_New_Condition		

Figure 58. UAN : Get_Condition

TASK: Select_Existing_Condition		
User Actions	**Feedback**	**System State**
~[select_button]; M_Lv∧	select_button ! -! Display(select_menu)	Select enabled
~[cond_name]; M_Lv∧	cond_name ! -! Remove(select_menu)	Conditon is selected

Figure 59. UAN : Select_Existing_Condition

TASK: Select_New_Condition		
User Actions	**Feedback**	**System State**
~[add_button]; M_Lv∧	add_button ! -! Remove(select_menu) Display(add_menu)	Add enabled
K(name); Rv∧	name is displayed	
K(description);	description is displayed	
~[base_selector]; M_Lv∧*	base′ is displayed	
~[ok_button]; M_Lv∧	ok_button ! -! Remove(add_menu)	Add condition expression
Add_Time_Condition | Add_State_Condition | Add_Mixed_Condition	Display(time_menu) | Display(state_menu) | Display(mixed_menu)	

Figure 60. UAN : Select_New_Condition

TASK: Add_Mixed_Condition		
User Actions	**Feedback**	**System State**
Add_State_Condition		
Add_Time_Condition		

Figure 61. UAN : Add_Mixed_Condition

TASK: Add_State_Condition		
User Actions	**Feedback**	**System State**
K(expression);	expression is displayed	
~[ok_button]; M_Lv∧	ok_button ! -!	Expression is accepted

Figure 62. UAN : Add_State_Condition

TASK: Add_Time_Condition		
User Actions	**Feedback**	**System State**
Add_New_Alarm | Select_Existing_Alarm		

Figure 63. UAN : Add_Time_Condition

TASK: Add_New_Alarm		
User Actions	**Feedback**	**System State**
~[add_button]; M_Lv∧	add_button ! -! Display(alarm_name_menu)	Add alarm enabled
~[alarm_name_menu]; K(alarm_name); Rv∧	Remove(alarm_name_menu)	Alarm entered

Figure 64. UAN : Add_New_Alarm

TASK: Select_Existing_Alarm		
User Actions	**Feedback**	**System State**
~[select_button]; M_Lv∧	select_button ! -! Display(select_menu)	Select enabled
~[alarm_name]; M_Lv∧	alarm_name ! -! Remove(select_menu)	Alarm is selected

Figure 65. UAN : Select_Existing_Alarm

TASK: Specify_Sti_Attribute		
User Actions	**Feedback**	**System State**
~[objects_icon]; $M_L v \wedge$	objects_icon ! -! Remove(spec_menu) Display(obj_menu)	Specify objects enabled
~[atts_icon]; $M_L v \wedge$	atts_icon ! -! Remove(obj_menu) Display(att_menu)	Specify attributes enabled
~[att_name]; $M_L v \wedge$	att_name ! -! Remove(att_menu) Display(type_menu)	
~[cstype_selector]; $M_L v \wedge *$	cstype′ is displayed	
~[range_selector]; $M_L v \wedge *$	range′ is displayed	
~[ok_button]; $M_L v \wedge$	ok_button ! -! Remove(type_menu) Display(sti_menu)	
Add_Value_Change | Delete_Value_Change | List_Value_Changes | Spec_Sti_Caps		

Figure 66. UAN : Specify_Sti_Attribute

TASK: Add_Value_Change		
User Actions	**Feedback**	**System State**
~[add_button]; M_Lv^	add_button ! -! Remove(sti_menu) Display(val_menu)	
Get_Value_Change		Value change added

Figure 67. UAN : Add_Value_Change

TASK: Delete_Value_Change		
User Actions	**Feedback**	**System State**
~[delete_button]; $M_L v \wedge$	delete_button ! -! Remove(sti_menu) Display(val_menu)	
Get_Value_Change		Value change deleted

Figure 68. UAN : Delete_Value_Change

TASK: List_Value_Changes		
User Actions	**Feedback**	**System State**
~[list_button]; M_Lv∧	list_button ! -! Display(val_list)	
~[ok_button]; M_Lv∧	ok_button ! -! Remove(val_list)	

Figure 69. UAN : List_Value_Change

TASK: Spec_Sti_Caps		
User Actions	**Feedback**	**System State**
~[specify_button]; M_Lv∧	specify_button ! -! Remove(sti_menu) Display(val_menu)	
Get_Value_Change	Display(cond_menu)	Current action: obj.attr := to_value
Get_Condition		

Figure 70. UAN : Spec_Sti_Caps

TASK: Get_Value_Change		
User Actions	**Feedback**	**System State**
K(from); Rv∧	from displayed	
K(to); Rv∧	to displayed	
~[ok_button]; M_Lv∧	ok_button ! -! Remove(val_menu) Display(sti_menu)	

Figure 71. UAN : Get_Value_Change

Function	*Description*
Display(x)	Cause menu x to be displayed on screen.
Remove(x)	Cause menu x to be removed from screen.

Figure 72. UAN : Function Definitions

Appendix B. Model Generator User's Manual

The Model Generator, its documentation, and associated utilites consist of approximately 20,000 lines of code in C and EQUEL-C under the SunView programming environment. This manual deals specifically with the tasks involved in defining and specifying a model using the Model Generator. For particular assisstance with using the SunView interface, refer to [Sun Microsystems 1988a, 1988b].

The Model Generator is comprised of two main executable processes, **mg3** and **modgen.** Mg3 is the main driver for the Model Generator; this program allows a modeler to select an existing model to work on, or create a new model, and guides the modeler in providing information required for new models. Mg3 invokes modgen, the model definition, specification, and documentation driver. To begin a session with the Model Generator, either call it from the SMDE window as shown in Figure 73, or invoke mg3 directly.

B.1 Create a Model

To create a new model, select the create icon on the driver menu. When creating a model, the modeler is prompted to enter the modeler's name, sponsor's name, a list of model definitions, a list of model assumptions, and a list of model objectives. This information is entered in the lower panel of the window. Note that the cursor must be in a panel to *activate* it, i.e. to direct keyboard input to that area of a window. An illustration of entering a model definition is presented in Figure 74.

When entering sponsor and modeler names, pressing the return key causes the text line to be accepted. When entering objectives, definitions, and assumptions, pressing return accepts the text and clears the line for the next entry. Press the escape key on a null line to quit entering information. (Note: pressing escape on a line that is not null results in the information on that line being lost. If this occurs, the line may be resubmitted by editing the appropriate list as described in Section B.6 of this User's Manual.)

B.2 Retrieve a Model

To retrieve an existing model to work on, select the retrieve icon on the driver menu. This brings up a list of all existing models. Select the model from the list by placing the mouse cursor on it, and depressing the left mouse button. A retrieval list is illustrated in Figure 75.

B.3 Exit the Model Generator

To exit the Model Generator, select the exit icon on the driver menu.

B.4 Define the Model

After selecting an existing model or creating a new one, the Model Generator invokes the definition/specification driver. This is shown in Figure 76. To define a model, select the define icon. This causes the model definiton menu to be displayed (as shown in Figure 77).

Note that the left section of the definition/specification driver contains a (hierarchical) list of all defined model objects. This list serves to drive the definition and specification processes. Select an object from the hierarchy to be the *current object,* i.e. the object for which model definition or specification is to be performed. The current object is highlighted in the list, and indicated on the left side of the definition/specification driver frame. A word of caution, change current objects only while at an outer-level menu (i.e. the main menu - Figure 76, the definition menu - Figure 77, or the specification menus - Figures 86 and 87). Changing the current object from other locations may result in unpredicatble system states.

B.4.1 Attach Attributes

To attach an attribute to the current object, select the button labeled "Attach." This causes the template for attribute definition to be displayed (as shown in Figure 78). Both the attribute name and description fields require text entry. These fields may be activated either by mouse selection

or alternately selected via repeatedly depressing the return key. The CM type and CM dimensionality fields are cyclic items. The current choice is displayed to the right of the label. Cycle through choices by clicking the left mouse button on the encircled arrows, or activate a pull-down menu by clicking the right mouse button on the choice string. Information is stored only if the button labeled "OK" is selected. To quit without saving, select "CANCEL." If the dimensionality is user defined, selecting "OK" causes a pop-up to be displayed in which the modeler enters the dimensionality. In this case, pressing return following dimensionality entry causes the information to be stored.

B.4.2 List Attributes

To list the attributes of the current object, select the button labeled "List." This causes a list of attributes to be displayed as illustrated in Figure 79. The list is scrollable, which allows viewing of all attributes even if they number larger than can be accommodated by the list display. Selecting "OK" will remove the list. The list may be left open (displayed) for as long as is convenient, however subsequent additions to the object's attribute set *will not* be reflected in the list.

B.4.3 Make a Subobject

To make a new object which will be directly subordinate to the current object, select the button labeled "Make." This causes a pop-up window to be displayed in which the object name is entered. Pressing return causes the text to be accepted, the pop-up removed, the model hierarchy updated and redisplayed with the new object as the current object. This scenario is depicted in Figure 80 and Figure 81.

B.4.4 Delete the Object

To delete the current object, select the button labeled "Delete." This causes an alert to be displayed which asks the modeler to reaffirm the delete selection. If the modeler responds in the affirmative, the object (its attributes and specification information, if any) is deleted. The hierarchy is updated and redisplayed, however the hierarchy is not renumbered.

The modeler is not permitted to delete (or modify the name of) the top level object.

B.4.5 Retrieve a Subobject

The facility to retrieve a subobject is not currently implemented in the Model Generator pending research into the Premodels Manager.

B.4.6 Create a Set

To create a set object which will be directly subordinate to the current object, select the button labeled "Create." This causes a pop-up window to be displayed in which the object name is entered. Pressing return causes the text to be accepted, the pop-up removed, the model hierarchy updated and redisplayed with the new set object as the current object. The modeler is prompted to provide set description, set type, and, if a p-set, the number of elements in the set. When the information is correct and complete, selecting "OK" causes it to be entered into the database, and the set definition menu to be displayed. Entering set information is shown in Figure 82. The set definition menu, which is displayed in place of the object definition menu when the current object is a set, is shown in Figure 83. The set definition menu provides the same functions as the object definition

menu with the exception of not allowing the modeler to define subobjects or sets for a set (the options "Make," "Retrieve," and "Create") since sets exist as leaf nodes of a model decomposition tree.

B.4.6 Modify Model Definition

To modify the definition of the current object, select the button labeled "Modify." This causes the modeler to be presented with a menu allowing the selection of modify object name, or modify object attributes (or cancel) as shown in Figure 84. Selecting "Name" results in the display of a pop-up in which the new model name is entered, similar to the one for "Make" a subobject. Pressing return after entering the new name causes the pop-up to be removed and the model hierarchy updated and redisplayed with the new object name. Selcting "Attributes" results in the attribute information template shown in Figure 85 to be displayed. This template functions the same as the one for "Attach" attributes previously presented.

Note that when modifying or deleting information at the model definition level, model specification is being affected. Displays are in place which indicate to the modeler the affects on the specification of the definition change(s), however these are not yet implemented pending the addition of expression parsing to the Model Generator.

B.5 Specify the Model

Model specification is initiated by selecting the specification icon from the model definition/specification driver. This selection causes the main specification menu (shown in Figure 86) to be displayed.

B.5.1 Attributes

Selecting "Objects" from the main specification menu yields the object specification menu illustrated in Figure 87. Selecting "Attributes" from this menu allows the modeler to specify conditions causing value changes for attributes of the current object as delineated by attribute type. When "Attributes" is selected, a pop-up containing a menu of selectable attributes of the current object is displayed as shown in Figure 88. The modeler may select an attribute from the list or cancel the operation. Selecting an attribute from the list causes the list to be removed and an attribute specification template to be displayed. The template requests (via cyclic requestors) CS typing and range typing information. When the information is correct and complete selecting the button marked "OK" enters the information. Otherwise, the modeler may select "CANCEL" to terminate the specification action. Note the template values default to the ones currently stored in the database for the attribute. Therefore when respecifying an attribute the modeler may simply enter "OK" without changing CS type and range information that has been previously provided. Range type of an attribute is either ranged or enumerated. A range type of "ranged" means that a low value and a high value are provided, and "enumerated" means that each value an attribute may take on is identified. When "OK" is selected, a pop-up requesting either range or enumeration information is displayed. For range values, pressing return accepts the value and prompts for the next one. For enumerated values (as is the case for model objectives, definitions, and assumptions) pressing escape on a null line causes the termination of the value list. The enumerating pop-up and the attribute template are illustrated in Figure 89. Subsequent specification is attribute type dependent and presented below.

Note: enumerated values are not overwritten if no values are entered, i.e. if escape is pressed for the first enumerated value. This allows a modeler to maintain (possibly large) enumerated lists without having to re-enter the information each time an attribute is specified. However, range information (low and high values) must be provided each time an attribute is specified.

B.5.1.1 Status Transitional Indicative

The menu for status transitional indicative attribute specification is shown in Figure 90. Selecting "Add" results in the display of the menu in Figure 91. The modeler must enter a "from" value and a "to" value and select "OK" to enter the value change, or the modeler may choose to "CANCEL" the operation. Selecting "OK" stores the information and returns the modeler to the previous menu. If the values in the value change do not match the values enumerated for the attribute, the modeler is alerted and allowed to re-enter the value change or override the alert (as needed for specifying "counting variables"). Selecting "Delete" brings up an identical menu to that of Figure 91, except that entering "OK" results in the value change being deleted from the database. Selecting "List" causes a list of value changes for the attribute to be displayed. This list is pictured in Figure 92. Selecting "Specify" displays a menu identical to that for "Add" and "Delete" except that selecting "OK" causes the value change to be searched for in the database (the modeler is alerted if the value change has not been defined) and the menu of Figure 93 to be displayed. At this point the modeler may "List" conditions causing the value change as illustrated in Figure 94, or "Add" a condition causing the value change. Adding conditions is described in Section B.5.2.

B.5.1.2 Temporal Transitional Indicative

The menu for temporal transitional indicative attribute specification is shown in Figure 95. This menu serves to drive the specification of all attributes having types other than status transitional indicative. For a temporal transitional indicative attribute, selecting "Specify" results in the display of the action menu pictured in Figure 96. The modeler may indicate that the attribute receives a value as a result of an assignment, function call, or input. If the selection is "Input," the modeler has specified the action INPUT(obj.att). If the selection is "Assignment" or "Function Call," the modeler must provide the text of the right-hand side of the expression or the function call in a

pop-up window. Pressing return causes the expression to be stored and the pop-up window removed. In all cases, the modeler is next prompted to provide the condition precipitating the action.

B.5.1.3 Permanent Indicative

The menu for permanent indicative attribute specification is shown in Figure 95. This menu serves to drive the specification of all attributes having types other than status transitional indicative. For a permanent indicative attribute, selecting "Specify" results in the display of the action menu pictured in Figure 96. The modeler may indicate that the attribute receives a value as a result of an assignment, function call, or input. If the selection is "Input," the modeler has specified the action (INPUT(obj.att)) and is prompted to provide the condition precipitating the action. If the selection is "Assignment" or "Function Call," the modeler must provide the text of the right-hand side of the expression or the function call in a pop-up window.

The condition causing assignment to a permanent indicative attribute must be either initialization or termination. The modeler provides this information as shown in Figure 97.

B.5.1.4 Relational

The menu for relational attribute specification is shown in Figure 95. This menu serves to drive the specification of all attributes having types other than status transitional indicative. For a relational (coordinate or hierarchical) attribute, selecting "Specify" results in the display of the action menu pictured in Figure 98. The modeler must provide the text of the right-hand side of an expression to be assigned as a value to the attribute. When the information is correct, selcting "OK" stores the action. Subsequently, the modeler must identify the condition precipitating the action.

B.5.1.5 Time-Based Signal

The menu for time-based signal attribute specification is shown in Figure 95. This menu serves to drive the specification of all attributes having types other than status transitional indicative. For a time-based signal, selecting "Specify" results in the display of the action menu pictured in Figure 99. The modeler must provide the text of the expression which determines when the alarm will "go off." Also, the modeler may provide a parameter for the SET ALARM action being specified. The mechanics of this menu are identical to those of similar menus presented previously. When the information is correct, selecting "OK" causes the action to be stored and the modeler prompted to identify the condition precipitating the SET ALARM action.

B.5.2 Conditions

The menu for supplying model conditions is shown in Figure 100. When identifying a condition (or conditions) precipitating a model action, the modeler may "Select" a previously specified condition, or "Add" (specify) a new model condition. When all conditions precipitating the model action are identified, the model may "Quit" to return to the previous specification process.

If "Select" is chosen, the pop-up menu of Figure 101 is displayed from which the modeler may select (by placing the mouse cursor on the condition name and depressing the left mouse button) an existing model condition. If "Add" is selected, the specify condition template is displayed as illustrated in Figure 102. For a new model condition the modeler must provide a condition name (which must be unique among other condition names), a description of the condition, and a base. When the condition information is correct and complete, selecting "OK" enters the information. The modeler may also "CANCEL" the operation without creating a new condition.

When the condition is specified, either by selecting an existing condition or adding new condition information, the CAP is presented to the modeler as shown in Figure 103. When the modeler is finished viewing the CAP, selecting "OK" returns the modeler to the supply condition menu, allowing the specification of the next condition (if any) for the current action.

The base of a condition may be either "time," "state," or "mixed." The specification requirements of each type of condition are discussed below.

B.5.2.1 State-Based

For a state-based condition, the modeler must provide a boolean expression. Currently expression parsing is not implemented in the Model Generator, so expression syntax is modeler-dependent. The menu for this expression entry is shown in Figure 104.

B.5.2.2 Time-Based

For a time-based condition, the modeler must provide an alarm for the WHEN ALARM condition expression. The menu for alarm provision is shown in Figure 105. The modeler may "Select" an existing alarm (in a manner identical to selecting an existing condition) or "Add" a new alarm by providing the alarm name in a pop-up window (also shown in Figure 105) with the usual input semantics. New alarms are automatically attached as attributes of the top level model object, so alarm names should be unique among all attribute names of the top level object.

The modeler is also allowed to provide a parameter for the WHEN ALARM expression as shown in Figure 106.

B.5.2.3 Mixed

For a mixed condition, the modeler must provide a boolean expression and an alarm for the AFTER ALARM condition expression. This scenario is simply a combination of the two described above.

B.5.3 Object Creation and Destruction

Selecting ″Creation″ or ″Destruction″ from the object specification menu results in the display of the menu illustrated in Figure 107. The modeler may indicate the need for an identifier and provide its value if required, or ″CANCEL″ the operation. Selecting ″OK″ causes the information to be stored (as the CREATE or DESTROY action) and the modeler prompted to supply a condition that causes the action.

B.5.4 Set Membership for an Object

Selecting ″Set″ from the object specification menu allows a modeler to select a d-set from a selectable list of model d-sets (not shown) and then specify the condition(s) causing the current object′s insertion into or deletion from the d-set.

B.5.5 Initialization

Selecting ″Initialization″ from the main specification menu results in the display of the initialization specification menu illustrated in Figure 108. When specifying the initialization condition, a modeler

may provide an initial value for system time, identify the distinguishing permanent indicative attribute for p-sets (to be used in the for-loop for the p-set object creation), specify attribute initializations, specify creation of model objects, and schedule alarms. These options are discussed in detail below.

B.5.5.1 System Time

Selecting "System Time" from the initialization specification menu displays the menu illustrated in Figure 109. The value entered for system time should be a constant (generally zero). Selecting "OK" adds the assignment action *system_time := val* to the initialization condition. "CANCEL" terminates the process without storing the action.

B.5.5.2 P-sets

When P-sets are created, a for-loop is generated for the initialization condition of the form:

```
FOR i := 1 to <number in p-set> DO
    CREATE( <p-set_object> )
END FOR
```

When specifying p-set initialization, a modeler must identify the permanent indicative attribute that distinguishes p-set objects, thereby forming the action: *<p-set_object>.<pi_att> := i* which is placed in the for-loop immediately following the CREATE statement.

Selecting "P-set" from the initialization specification menu causes the pop-up window pictured in Figure 110 to be displayed, but only if the current object is a p-set. If the current object is not a p-set, the modeler is alerted and may change the current object by selecting an object from the hierarchy. If the current object is a p-set, the pop-up window allows the modeler to select from the list of attributes of the p-set the permanent indicative attribute that distinguishes set objects. If the

attribute selected is not a permanent indicative attribute, the modeler is alerted and allowed to re-select.

B.5.6 Termination

Selecting "Termination" from the main specification menu results in the display of the termination specification menu illustrated in Figure 111. When specifying the termination condition, a modeler may provide the expression for the condition, specify attribute assignment, and specify model output. These options are discussed in detail below.

Selecting "Expression" from the termination specification menu displays the menu illustrated in Figure 112. The expression for termination should be boolean (and is currently unparsed). Selecting "OK" updates the termination condition. "CANCEL" terminates the process without storing the expression.

B.5.7 Model Input and Output

Selecting "Input/Output" from the main specification menu causes a pop-up menu containing a list of attributes of the current object to be displayed (the same menu as in Figure 88). After selecting an attribute of the current object, the menu of Figure 113 is displayed. The modeler may select "Input" or "Output" and is then prompted to indentify a condition precipitating the action.

B.5.8 Functions

The facility for function specification is not currently implemented in full pending further research into the Condition Specification's role in function specification. Currently, selecting "Functions" from the model specification menu allows a modeler to provide function typing and parameter typing (not shown).

B.5.9 Monitored Routines

Selecting "Monitored" from the model specification menu allows the modeler to select an attribute of the current object an indicate its monitoring as any combination of input monitoring, output monitoring, left monitoring, right monitoring or none (not shown).

B.5.10 Modify Model Specification

The facility for modifying a model specification is not fully implemented pending the addition of expression parsing within the Model Generator. Currently, selecting "Modify" from the model specification menu allows a modeler to delete a model condition or a model CAP.

B.6 List and Edit Features

The upper panel of the definition/specification driver contains buttons which provide a modeler access to important model information at all times during the definition/specification process. Se-

lecting any of the lists causes a pop-up window to be displayed. The window is scrollable, and may be left active for as long as desired. When finished viewing the window's contents, selecting "OK" causes the window to be removed. Figure 114 illustrates a listing of action clusters. Selecting any of the edits also causes a pop-up window to be displayed (see Figure 115). A modeler is allowed to add an item to a model list, delete an item from a list, or edit a list item. Figure 116 illustrates an list item edit.

Note: List edits are not reflected in the edit window. The edit window should be closed, and the appropriate list selected to view the update.

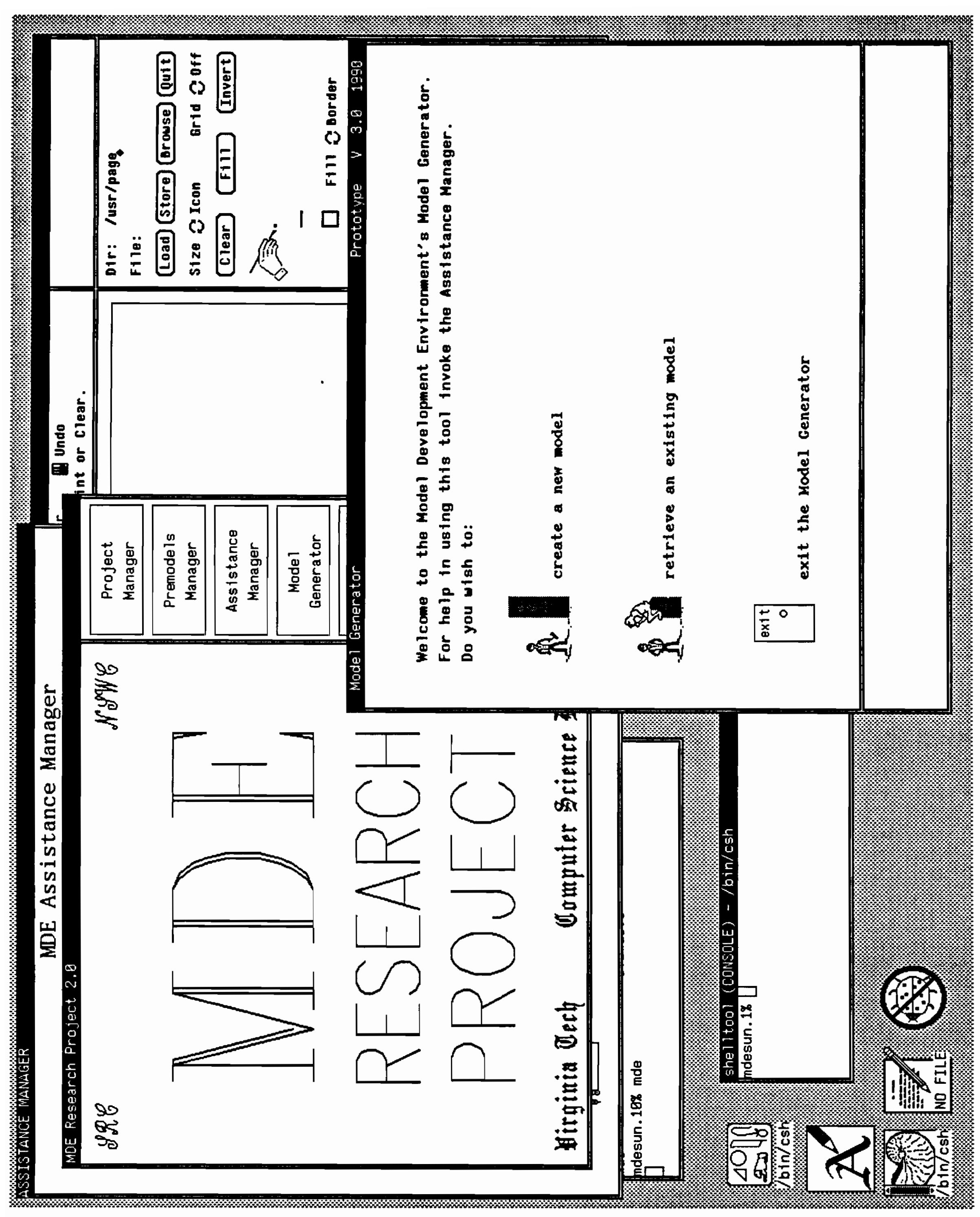

Figure 73. MG : The Model Generator in the Simulation Model Development Environment.

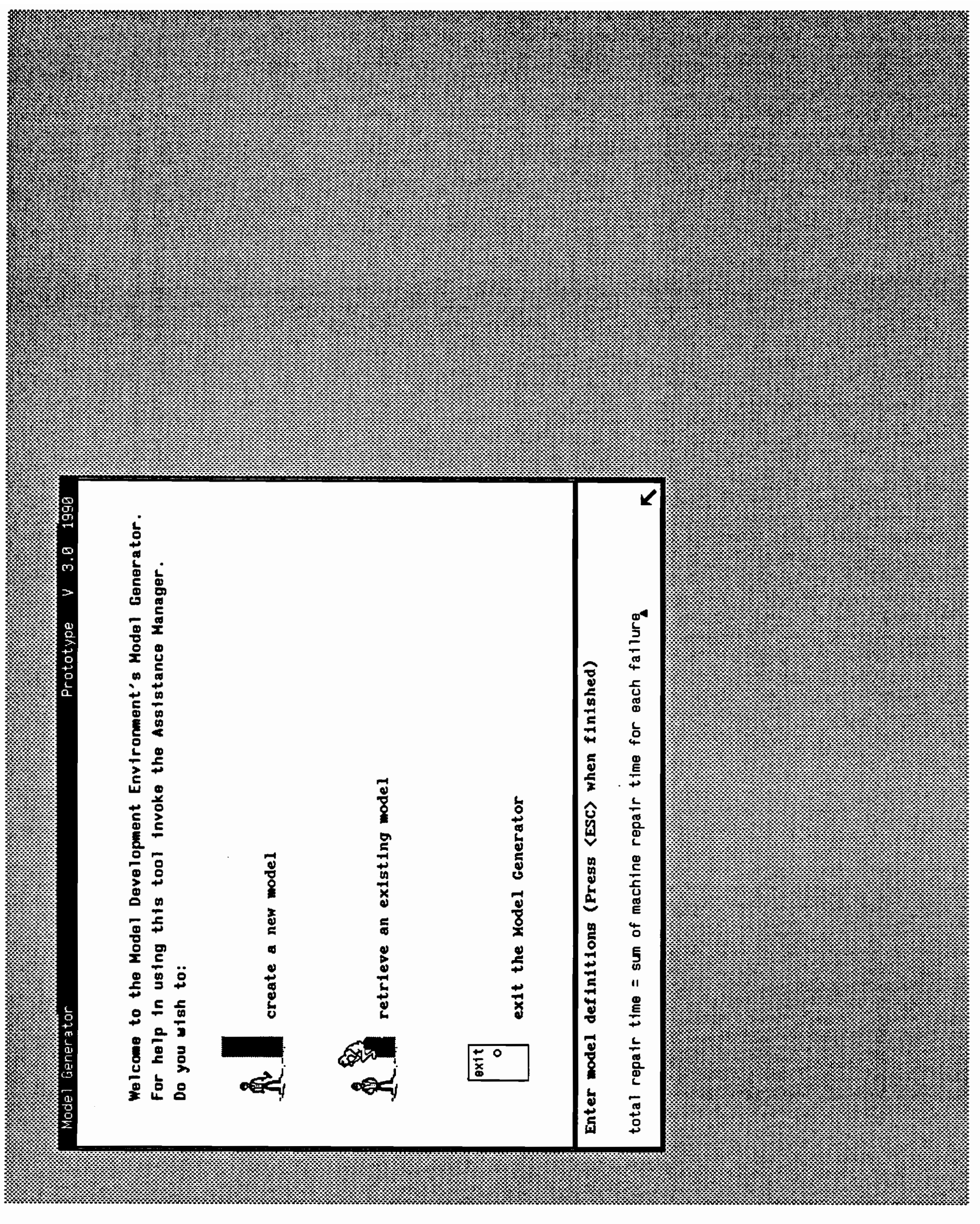

Figure 74. MG : Entering Model Definitions.

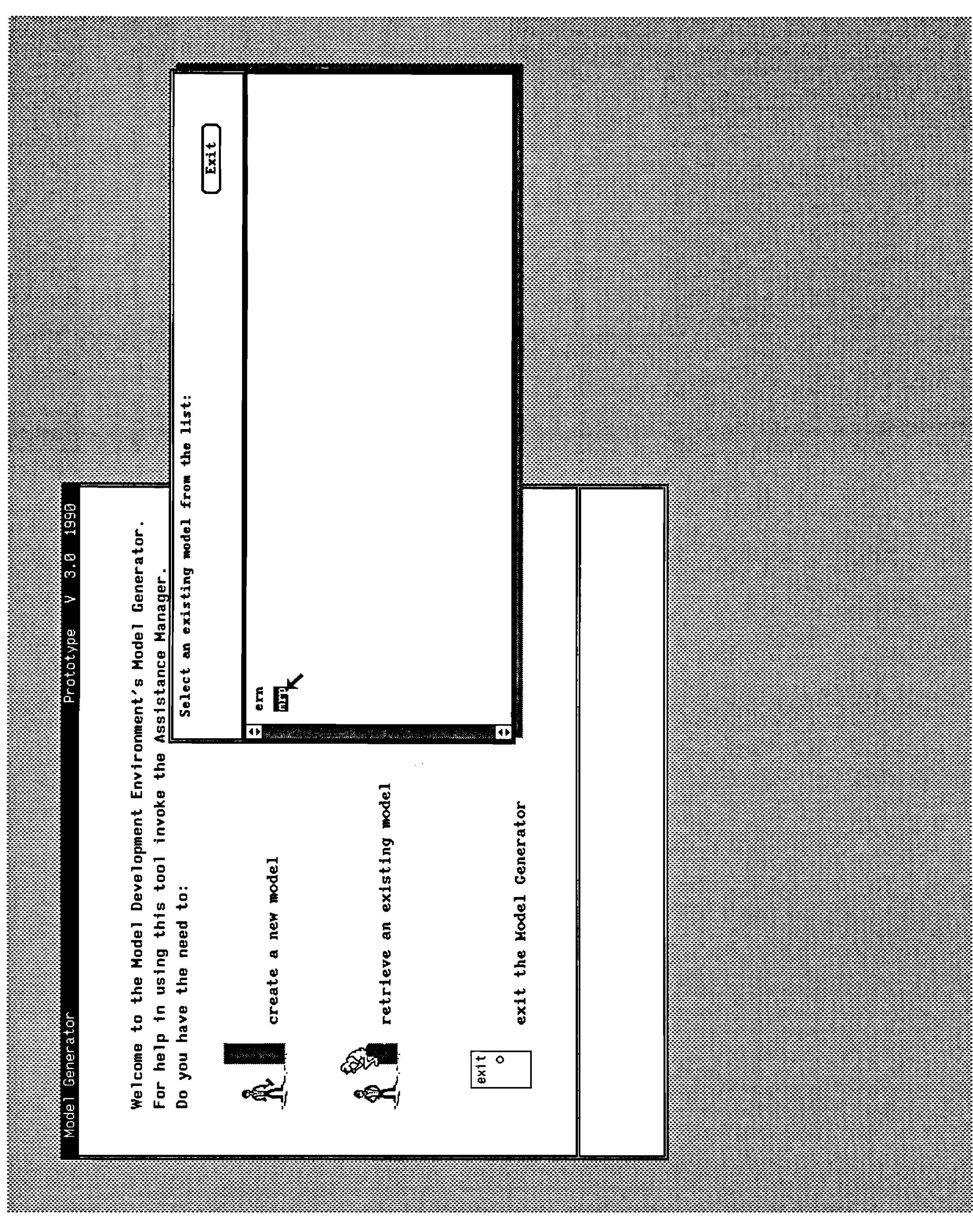

Figure 75. MG : Retrieving Existing Models.

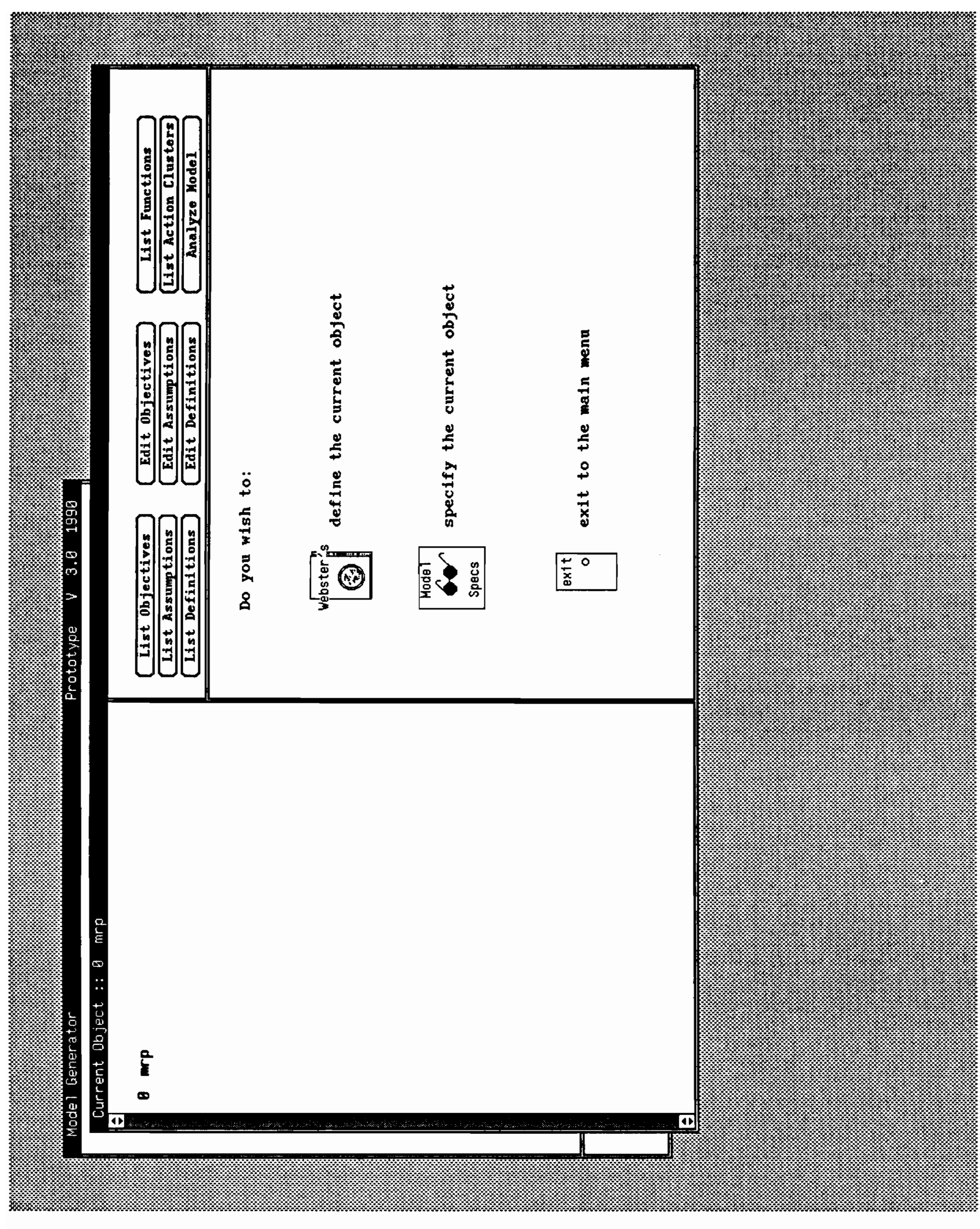

Figure 76. MG : The Definition/Specification Driver.

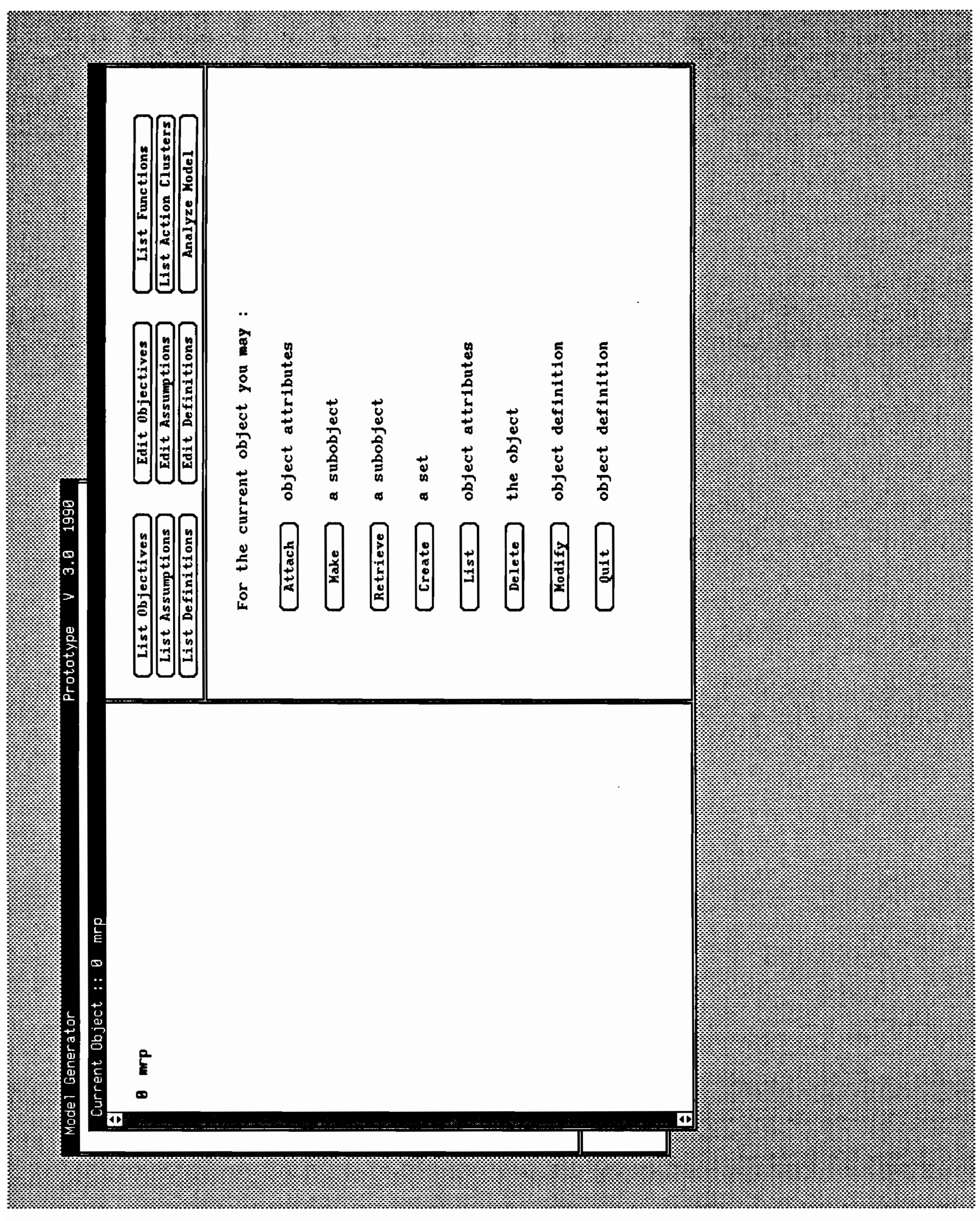

Figure 77. MG : Model Definition Menu.

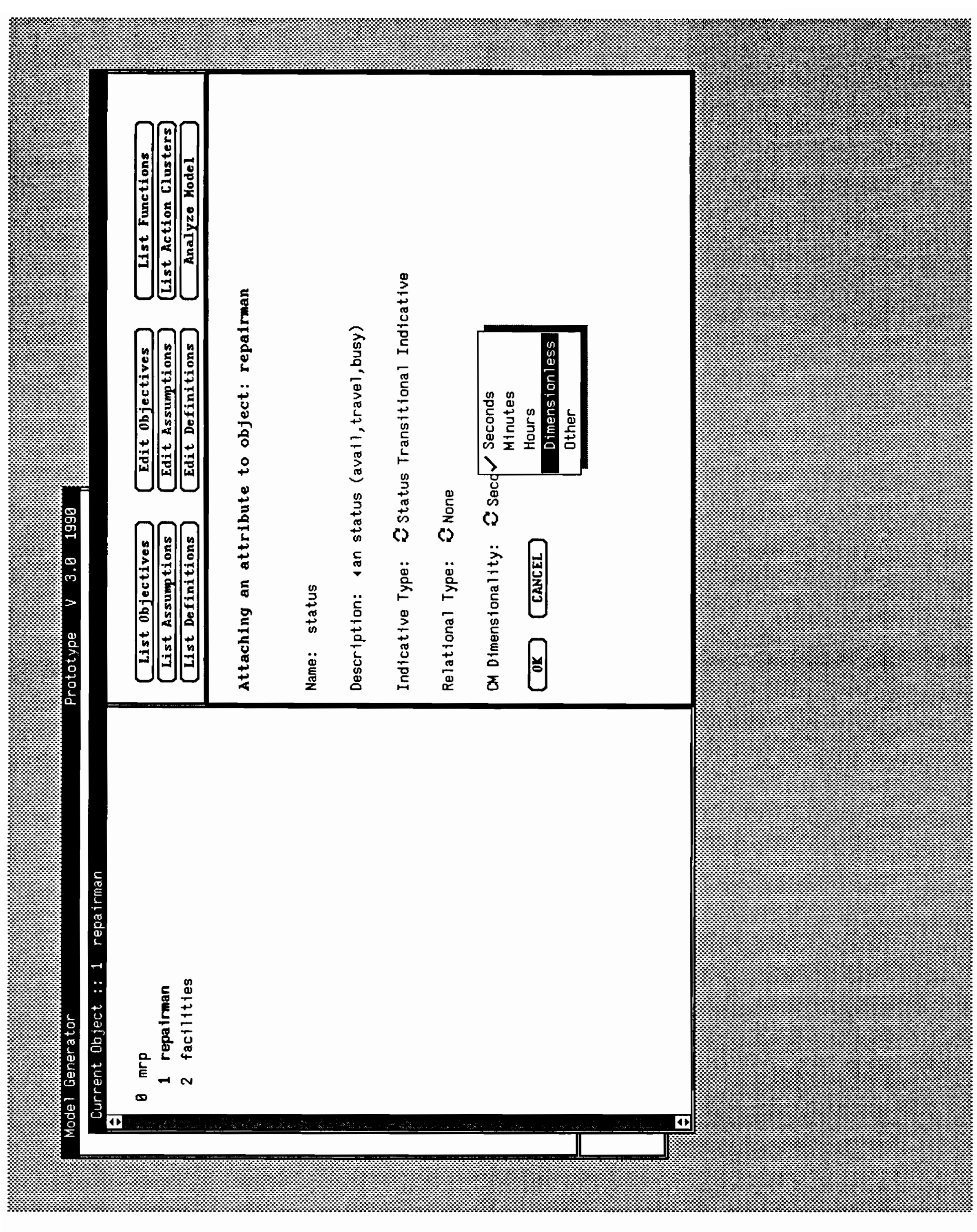

Figure 78. MG : Attaching Object Attributes.

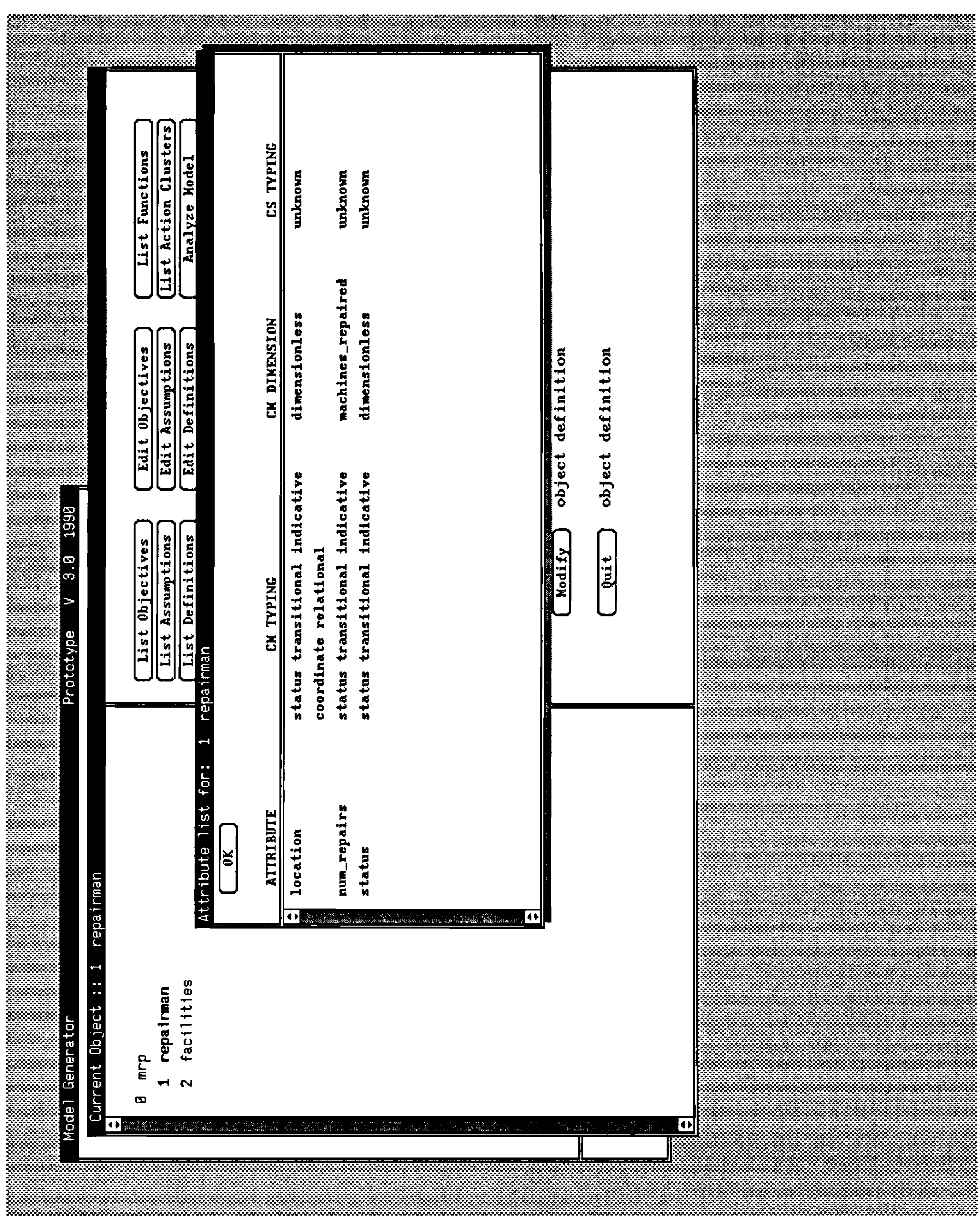

Figure 79. MG : Listing Object Attributes.

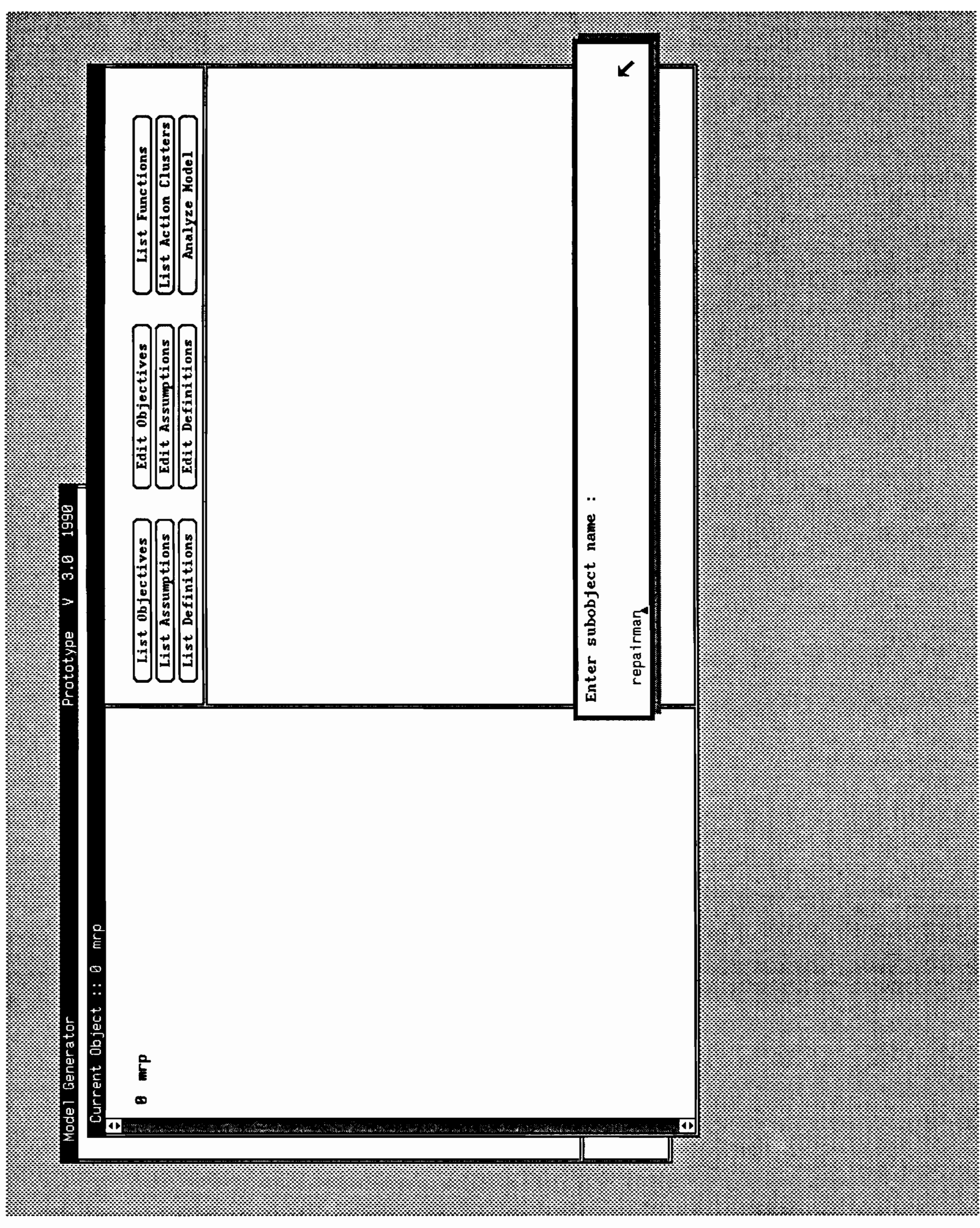

Figure 80. MG : Entering New Model Object.

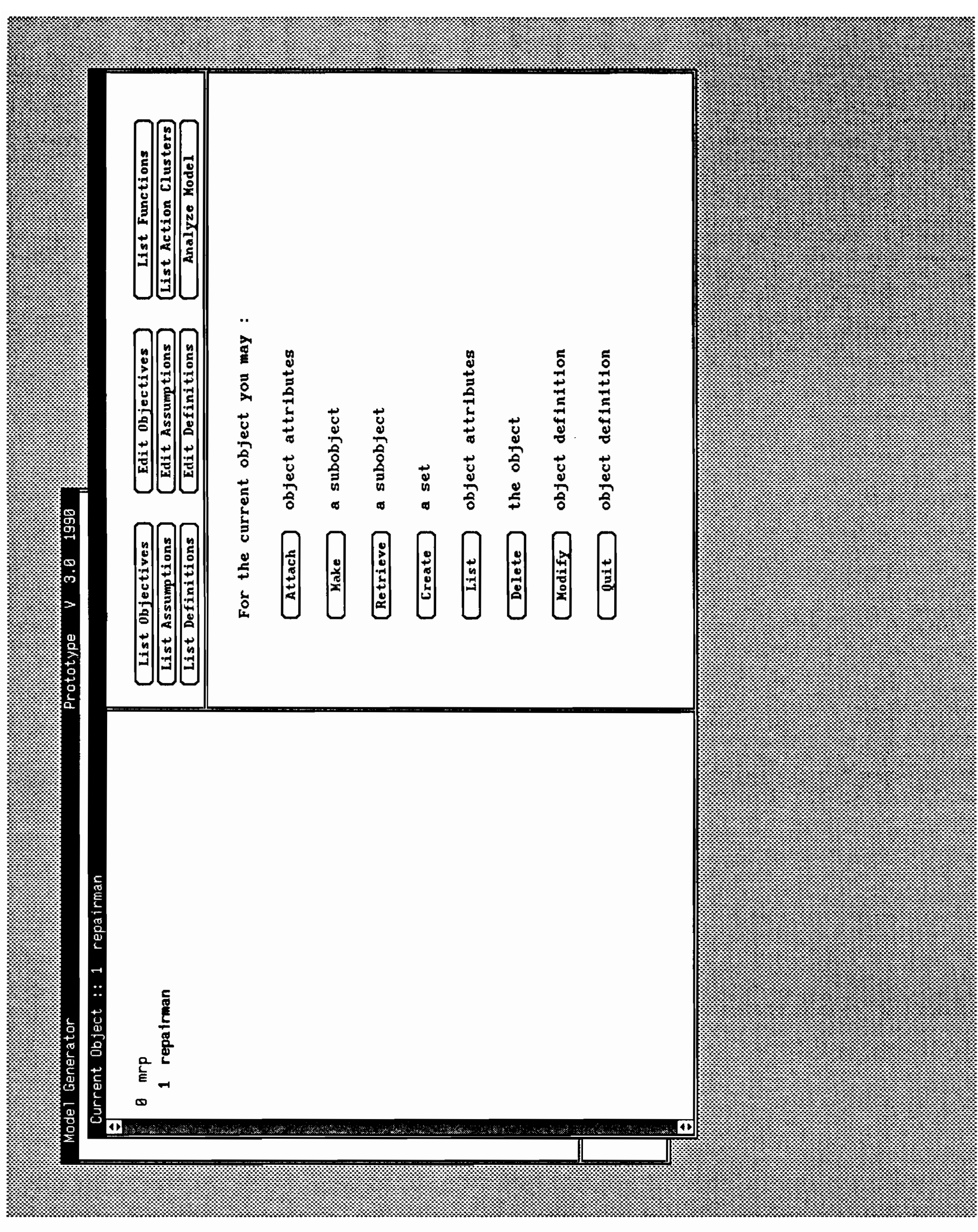

Figure 81. MG : New Object Becomes Current Object.

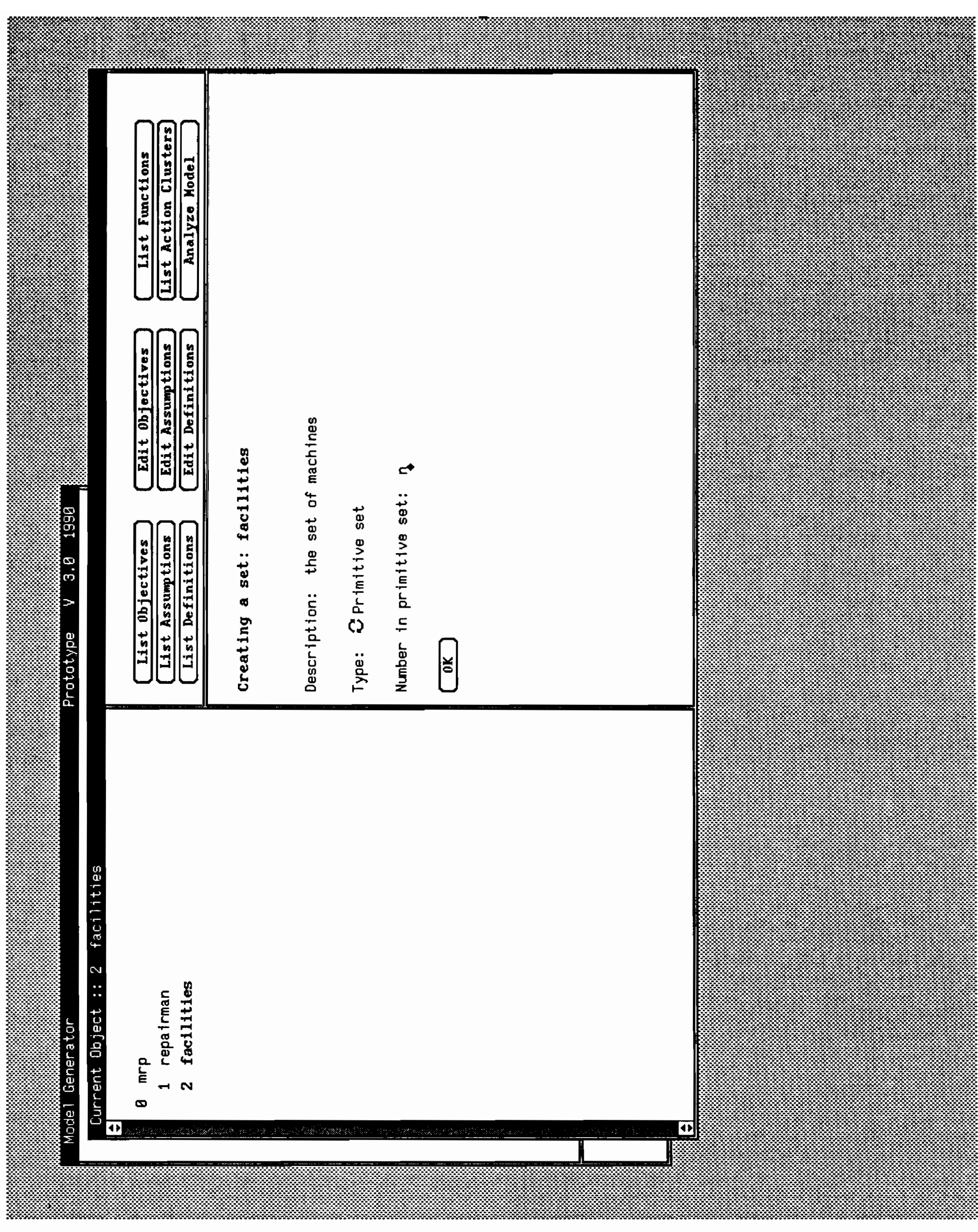

Figure 82. MG : Entering Set Information.

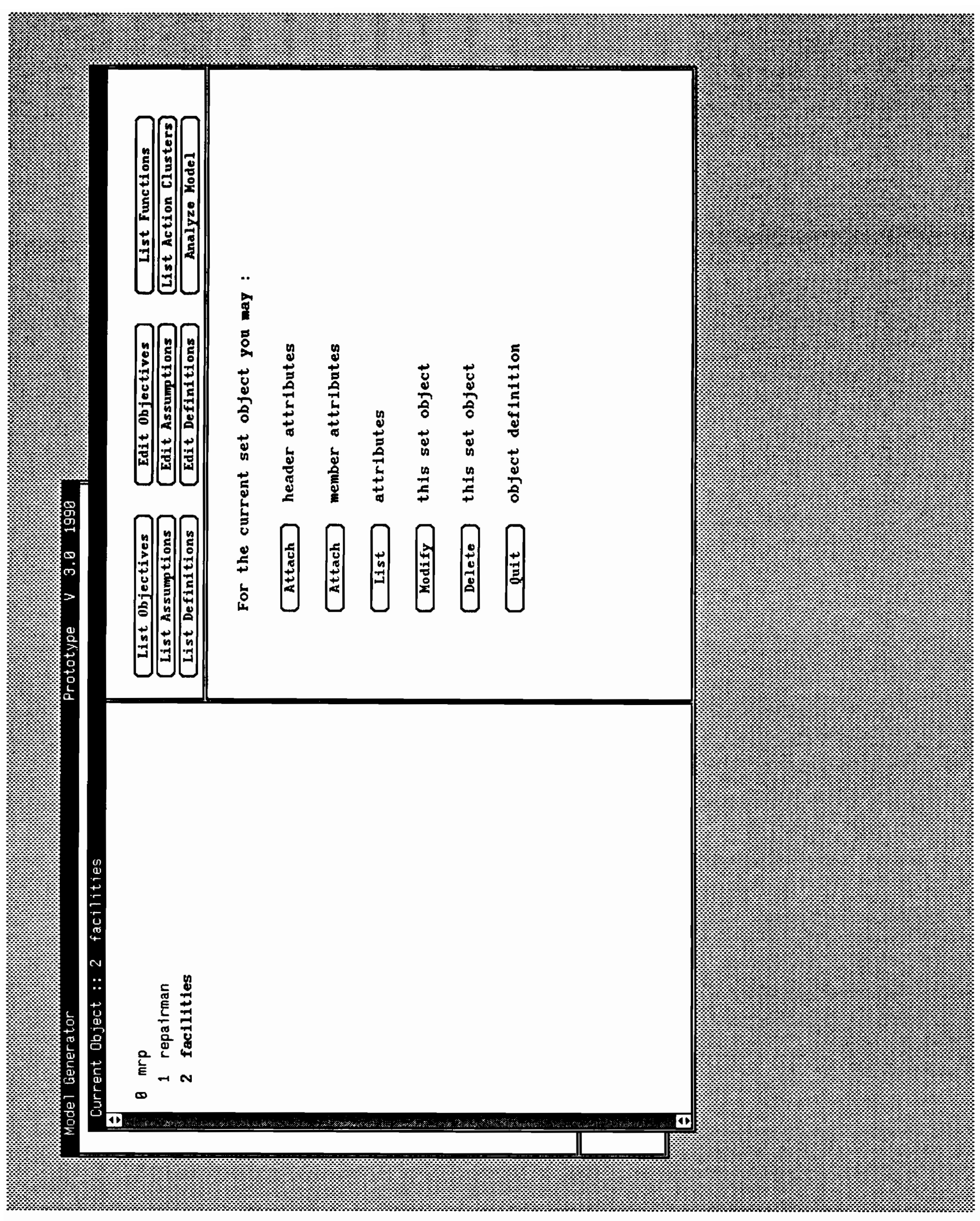

Figure 83. MG : Set Definition Menu.

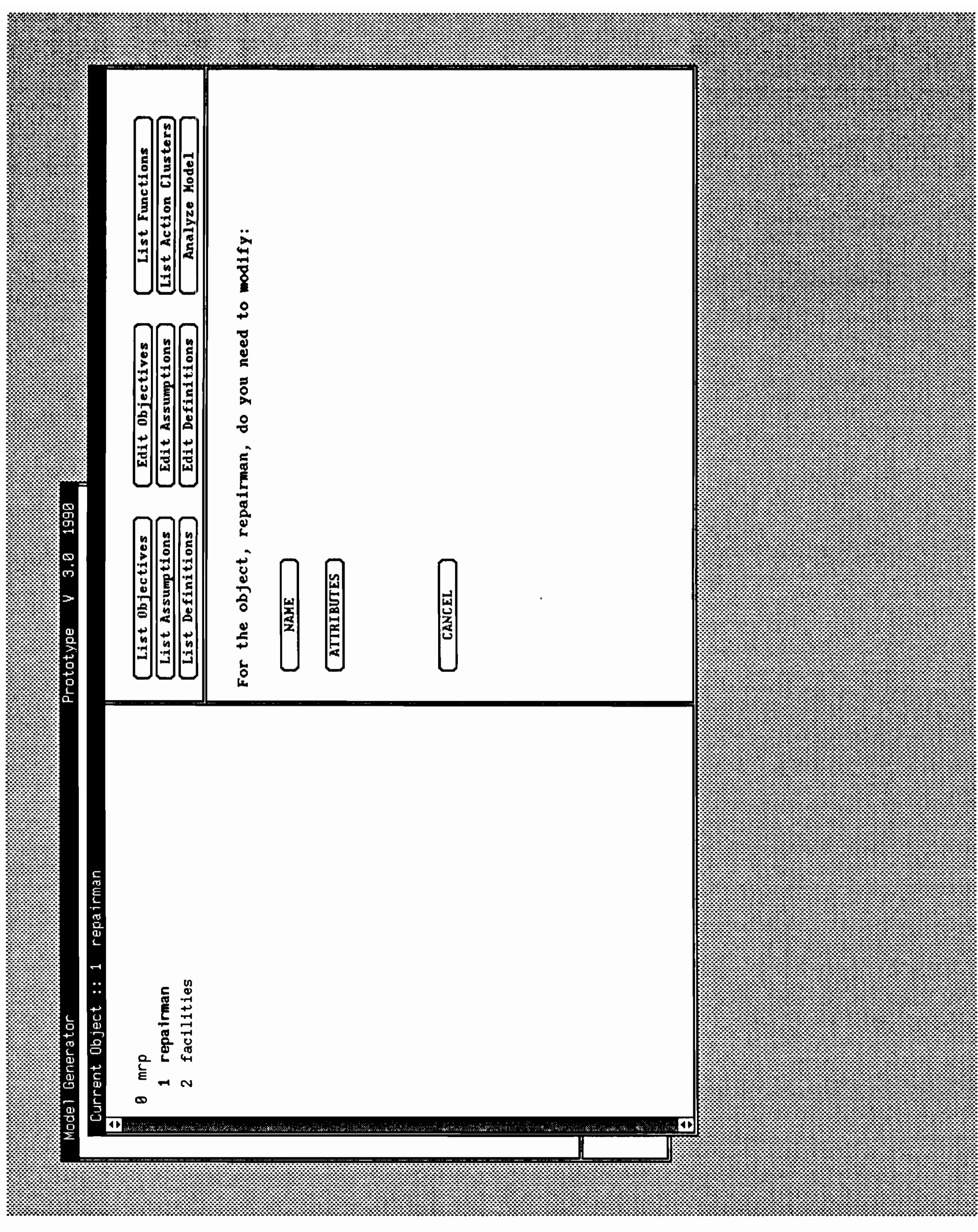

Figure 84. MG : Modify Object Menu.

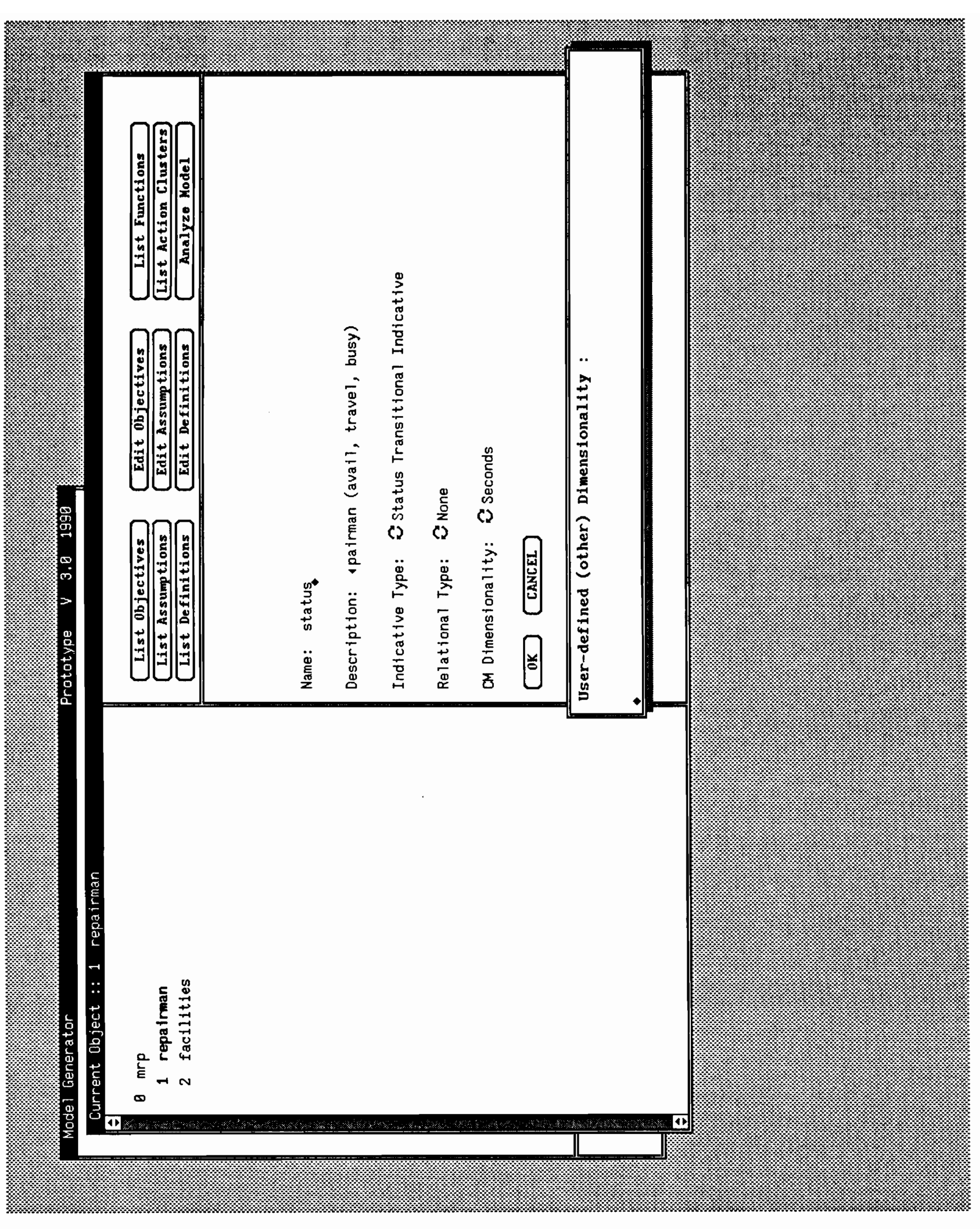

Figure 85. MG : Modifying Attribute Definition.

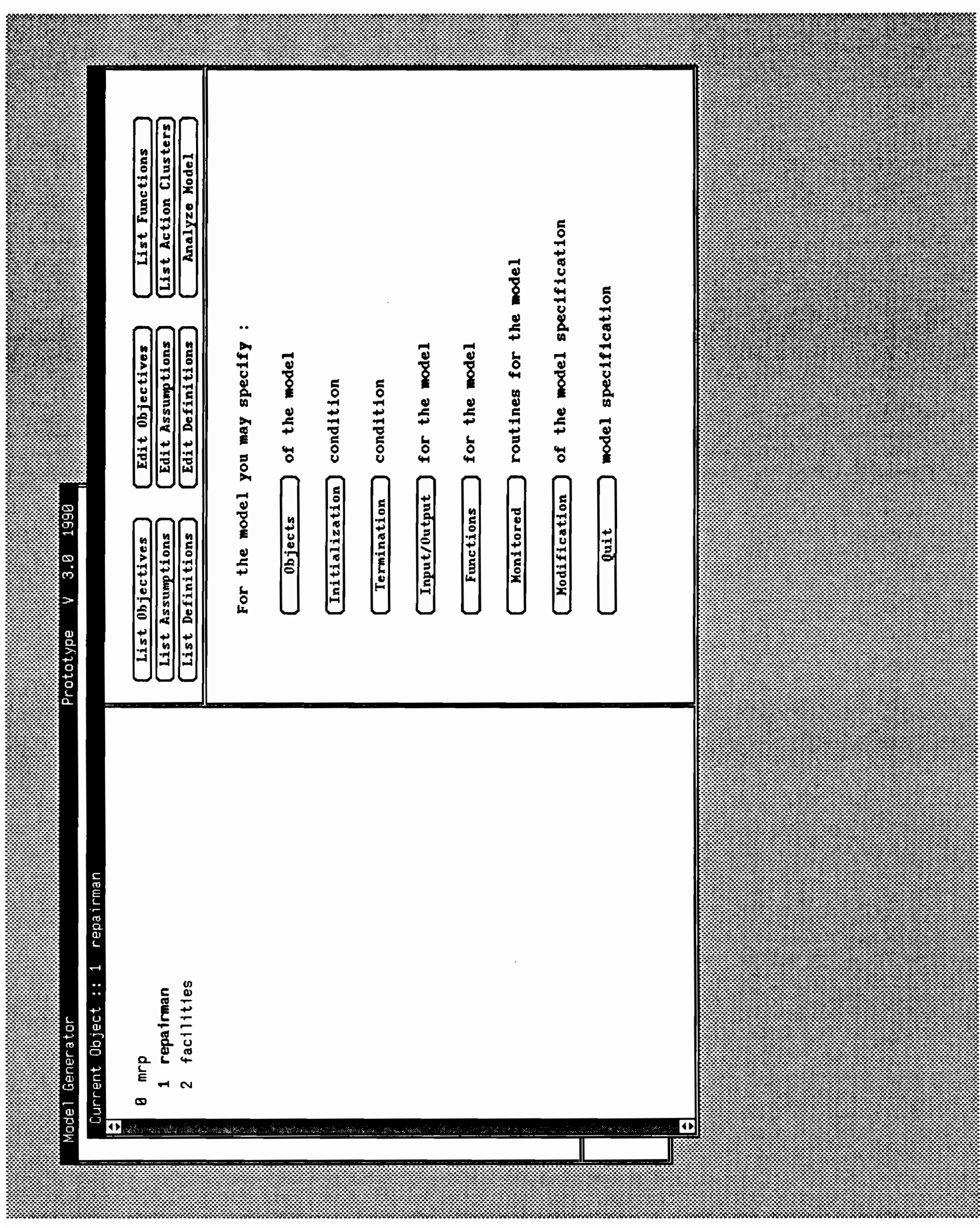

Figure 86. MG : Model Specification Menu.

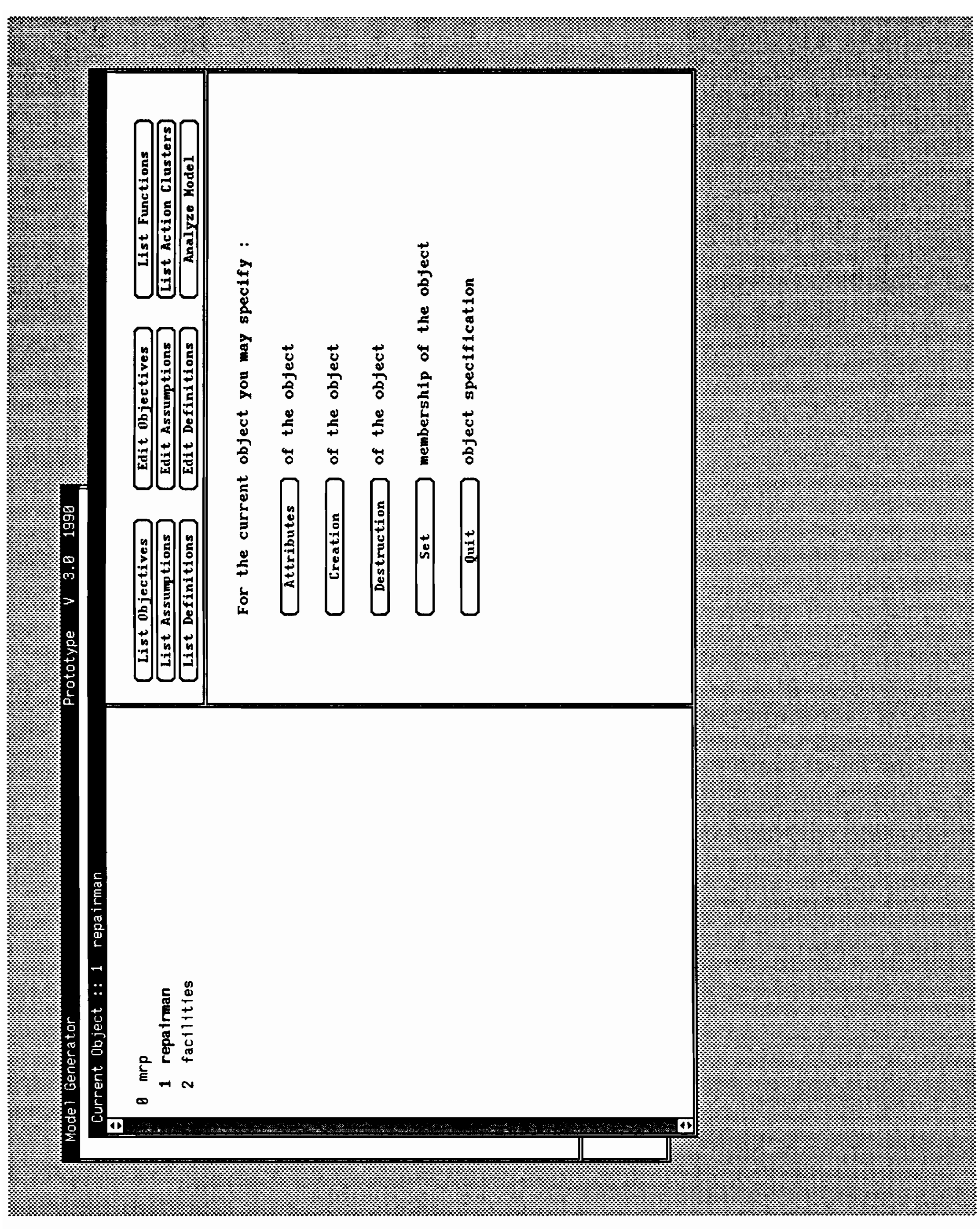

Figure 87. MG : Object Specification Menu.

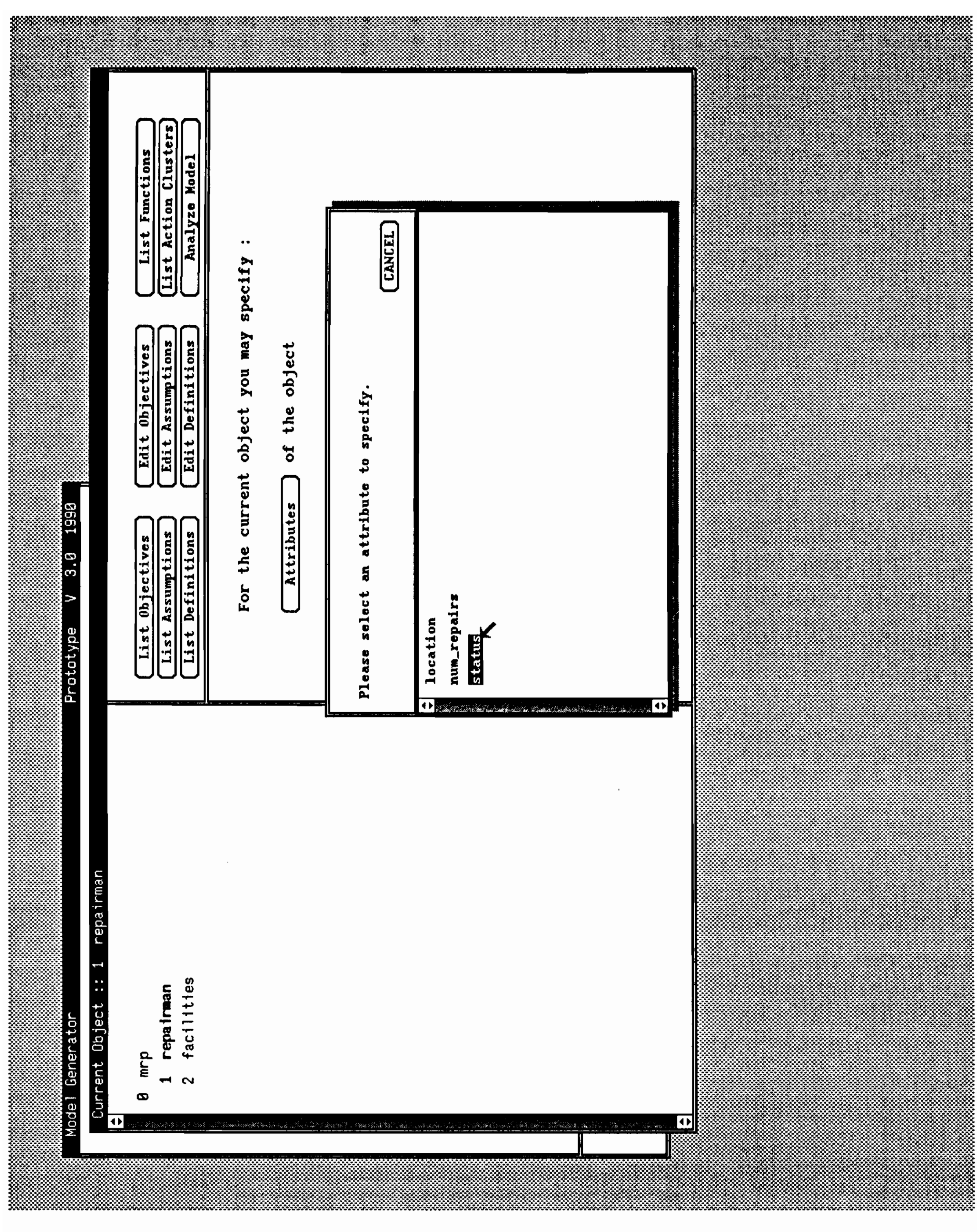

Figure 88. MG : Selecting Attribute to Specify.

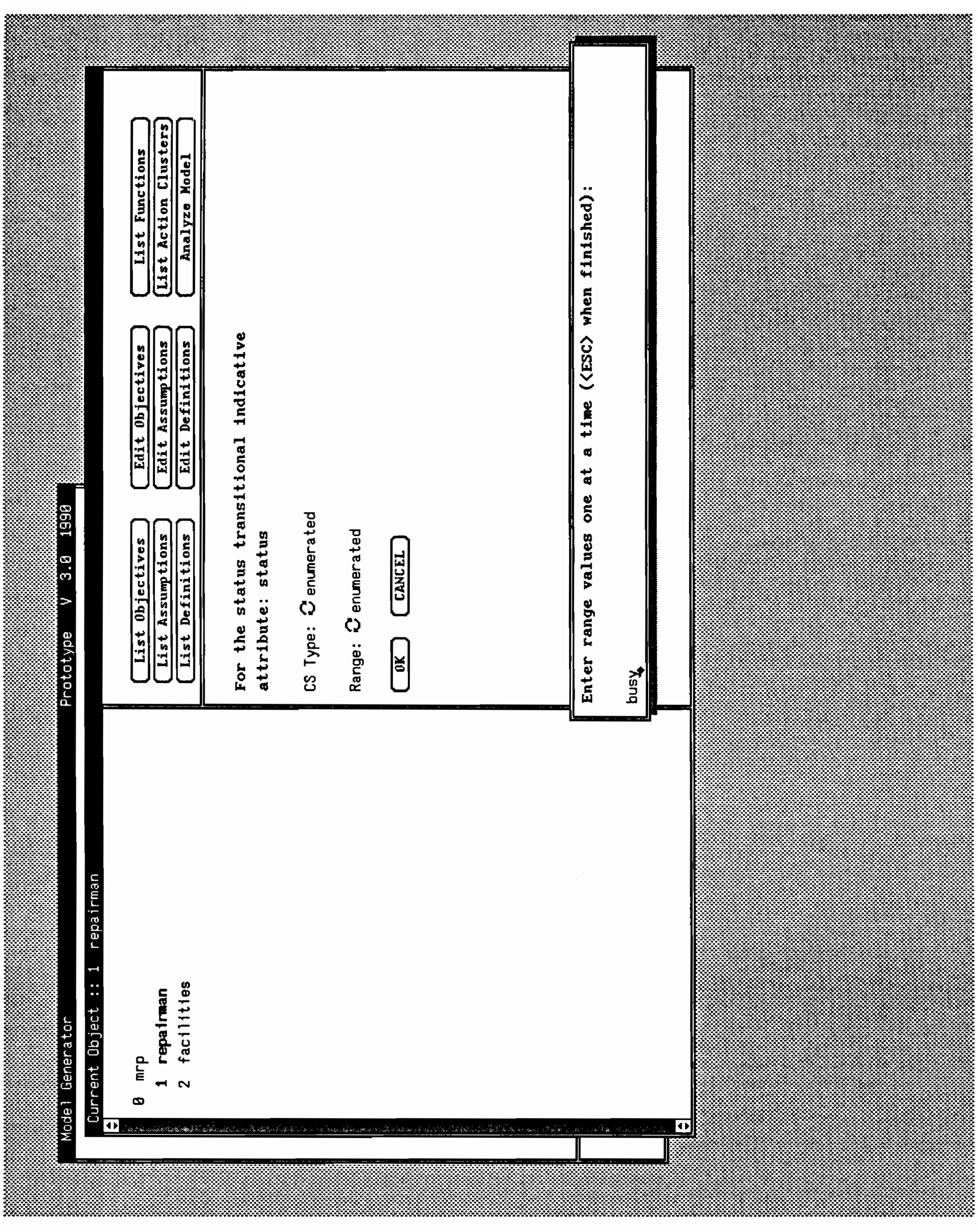

Figure 89. MG : Enumerating the Values of Status Transitional Indicative Attributes.

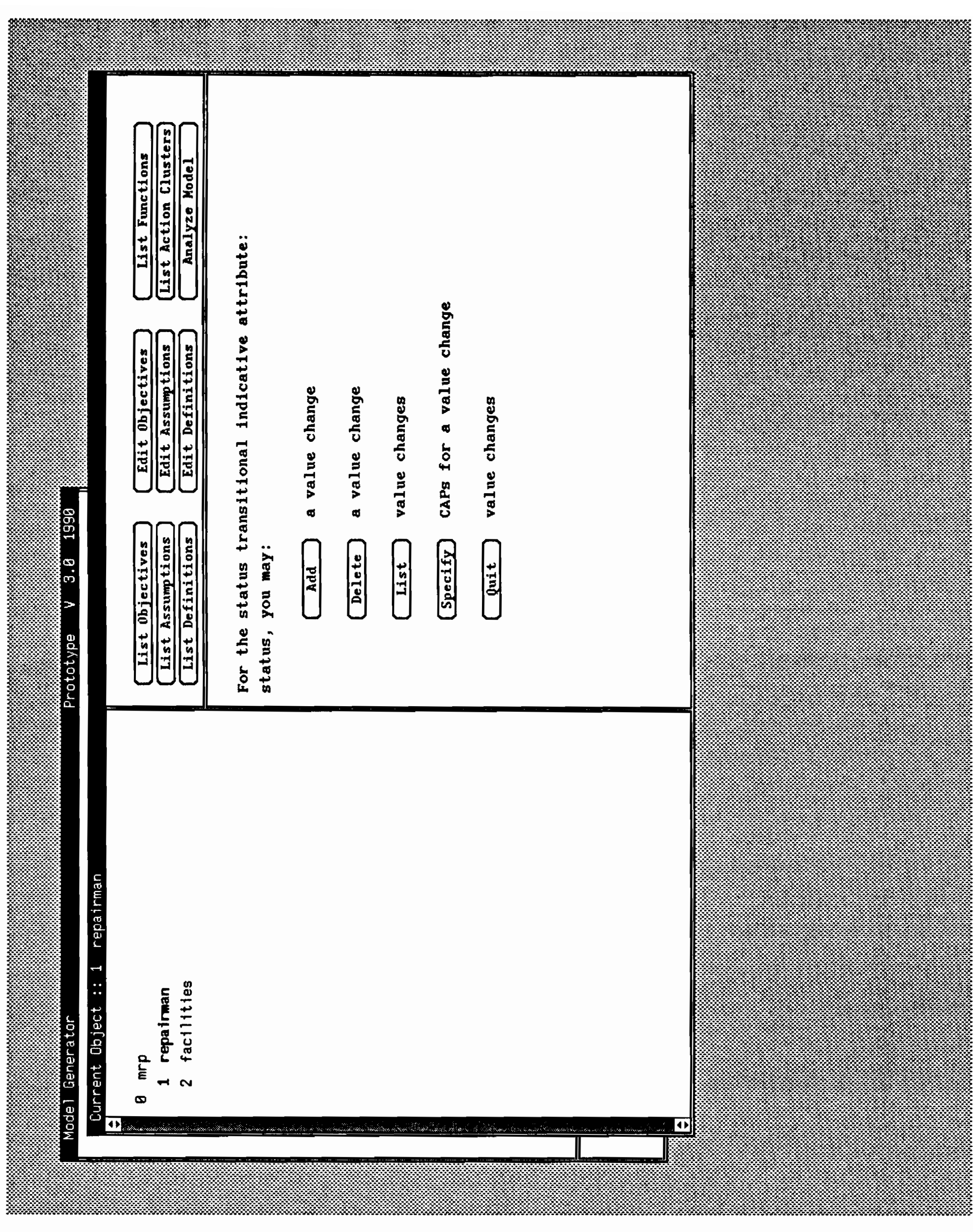

Figure 90. MG : The Status Transitional Indicative Attribute Specification Menu.

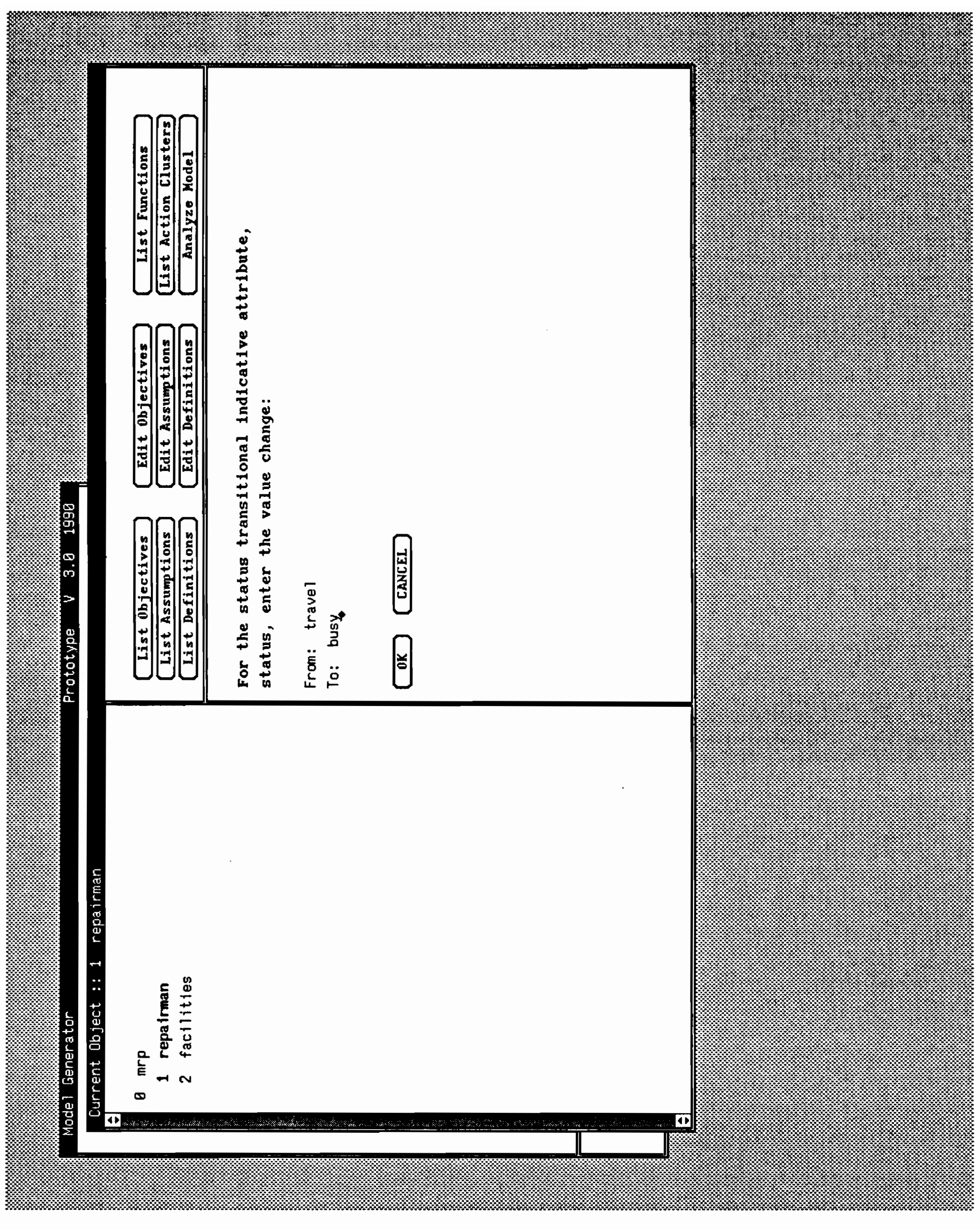

Figure 91. MG : Adding Value Changes for Status Transitional Indicative Attributes.

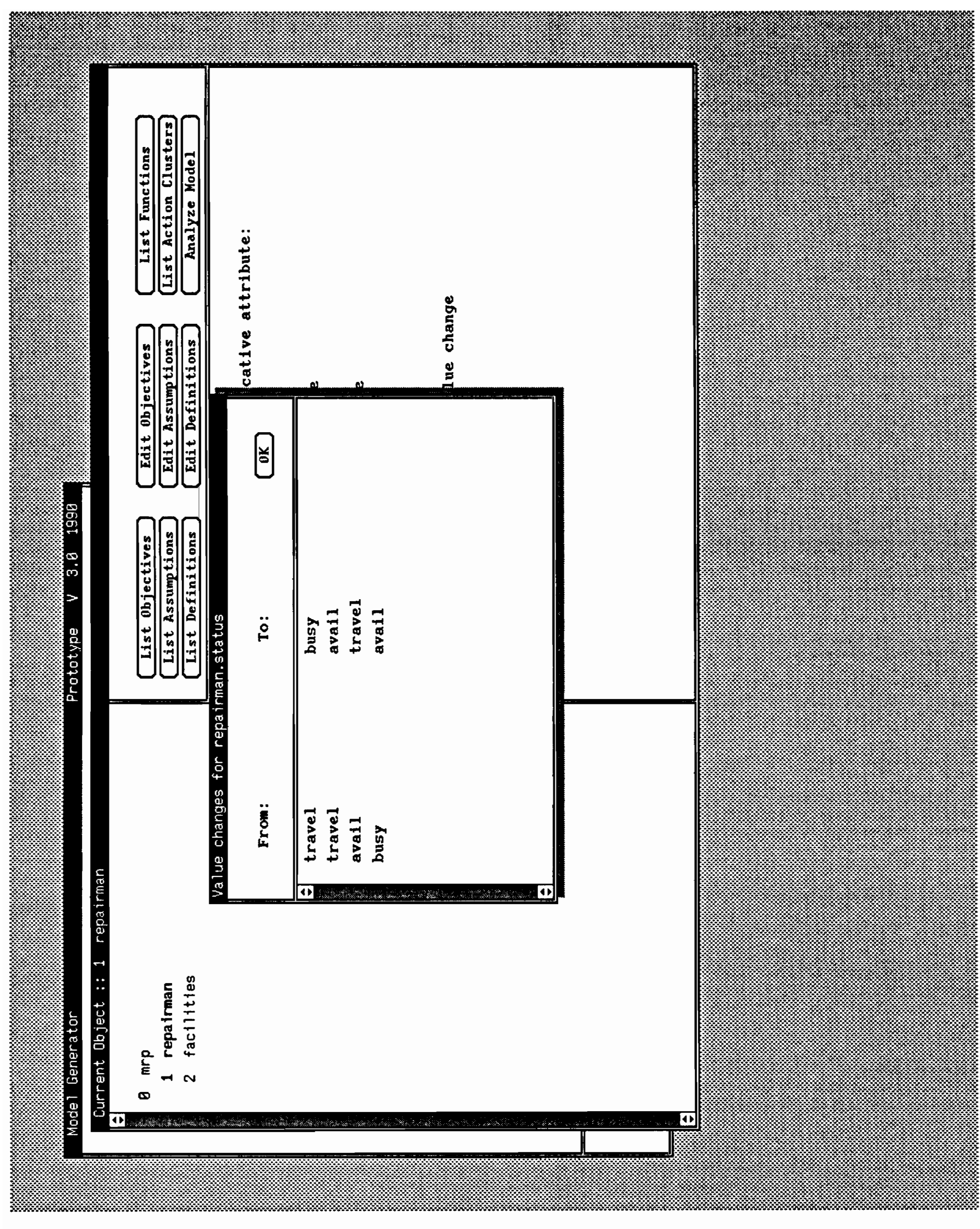

Figure 92. MG : Listing Value Changes for Status Transitional Indicative Attributes.

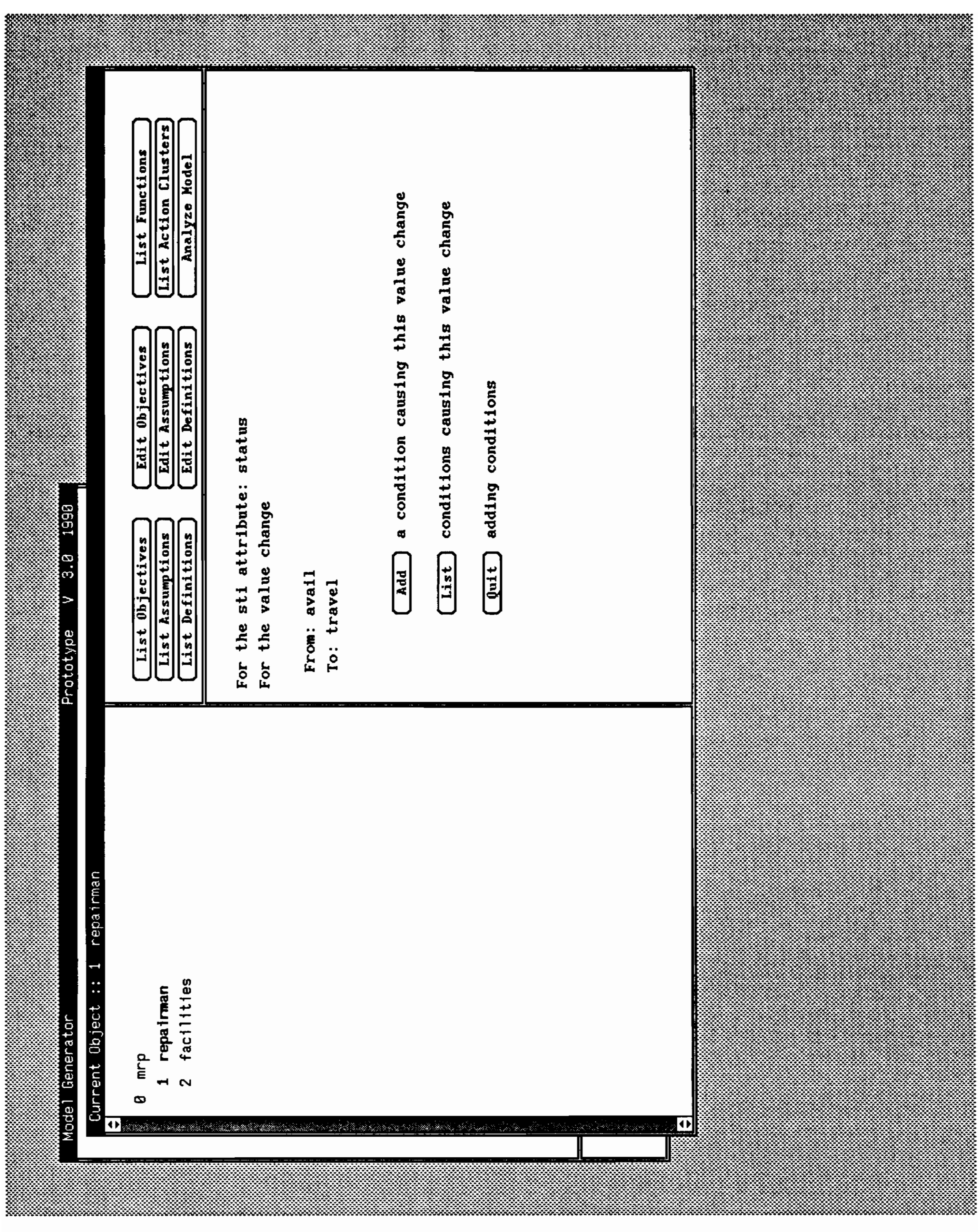

Figure 93. MG : Specifying Conditions for Change in Value for Status Transitional Indicative Attributes.

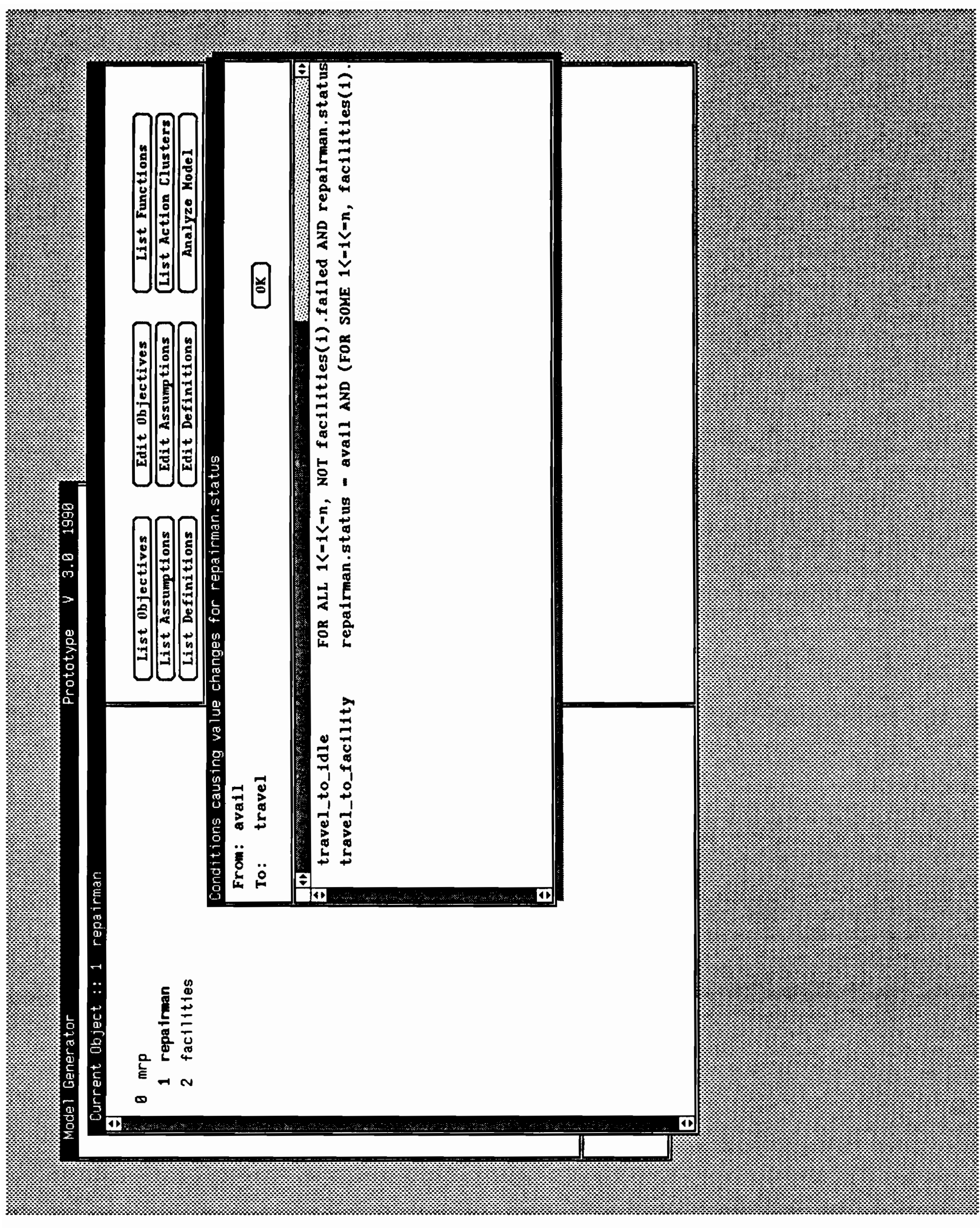

Figure 94. MG : Listing Conditions Causing Value Change for Status Transitional Indicative Attributes.

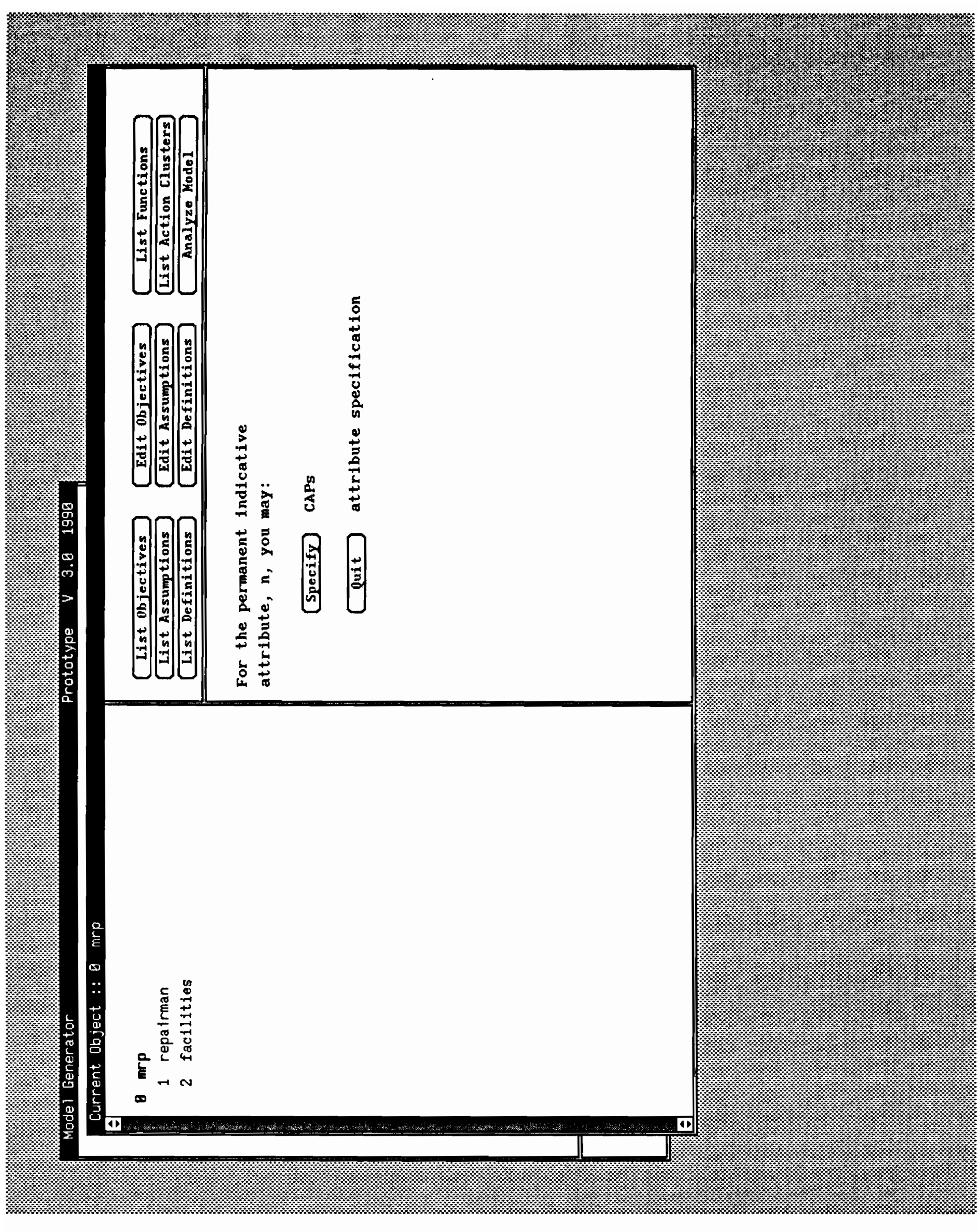

Figure 95. MG : Non-status Attribute Specification Menu.

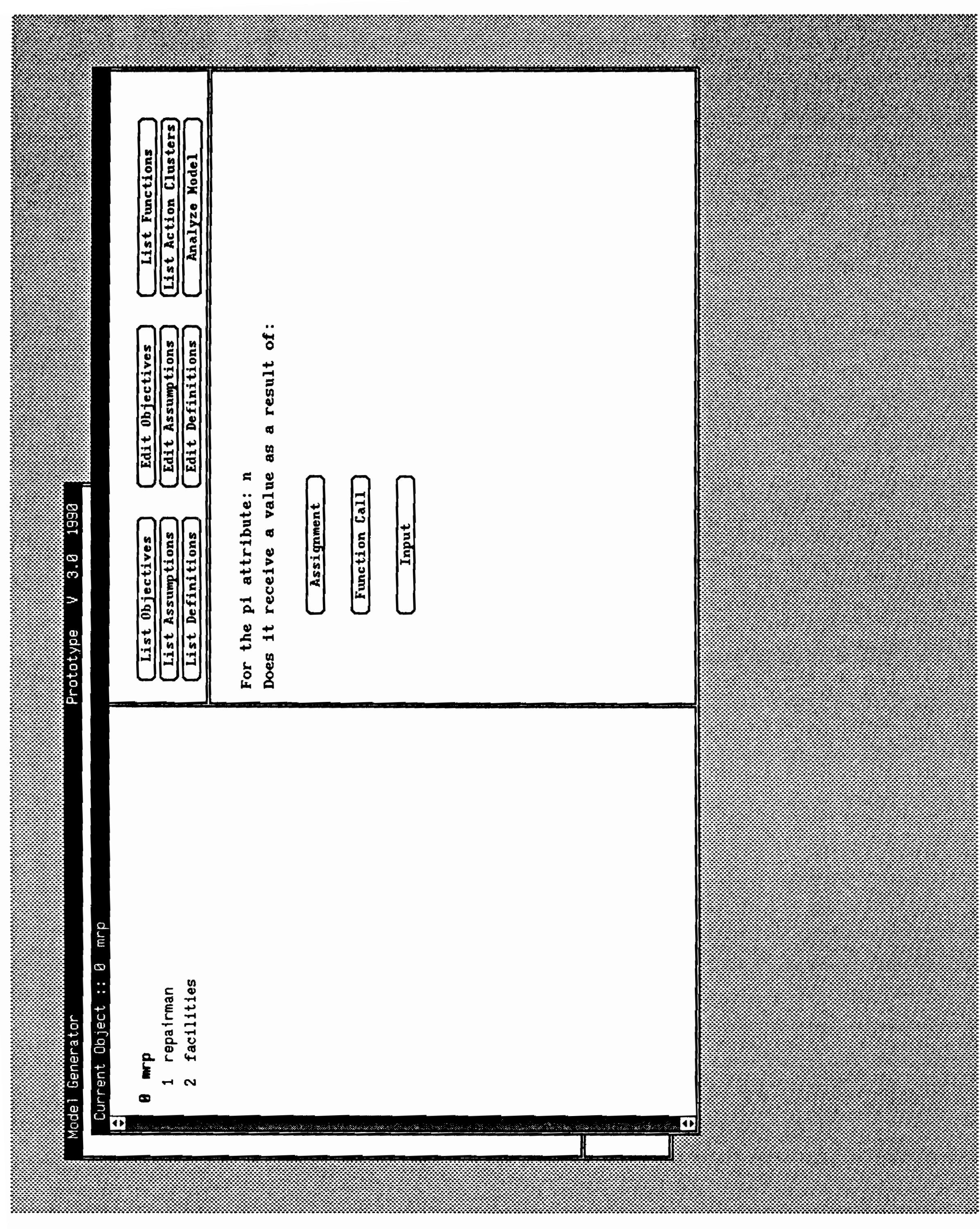

Figure 96. MG : Action Menu for Temporal and Permanent Attributes.

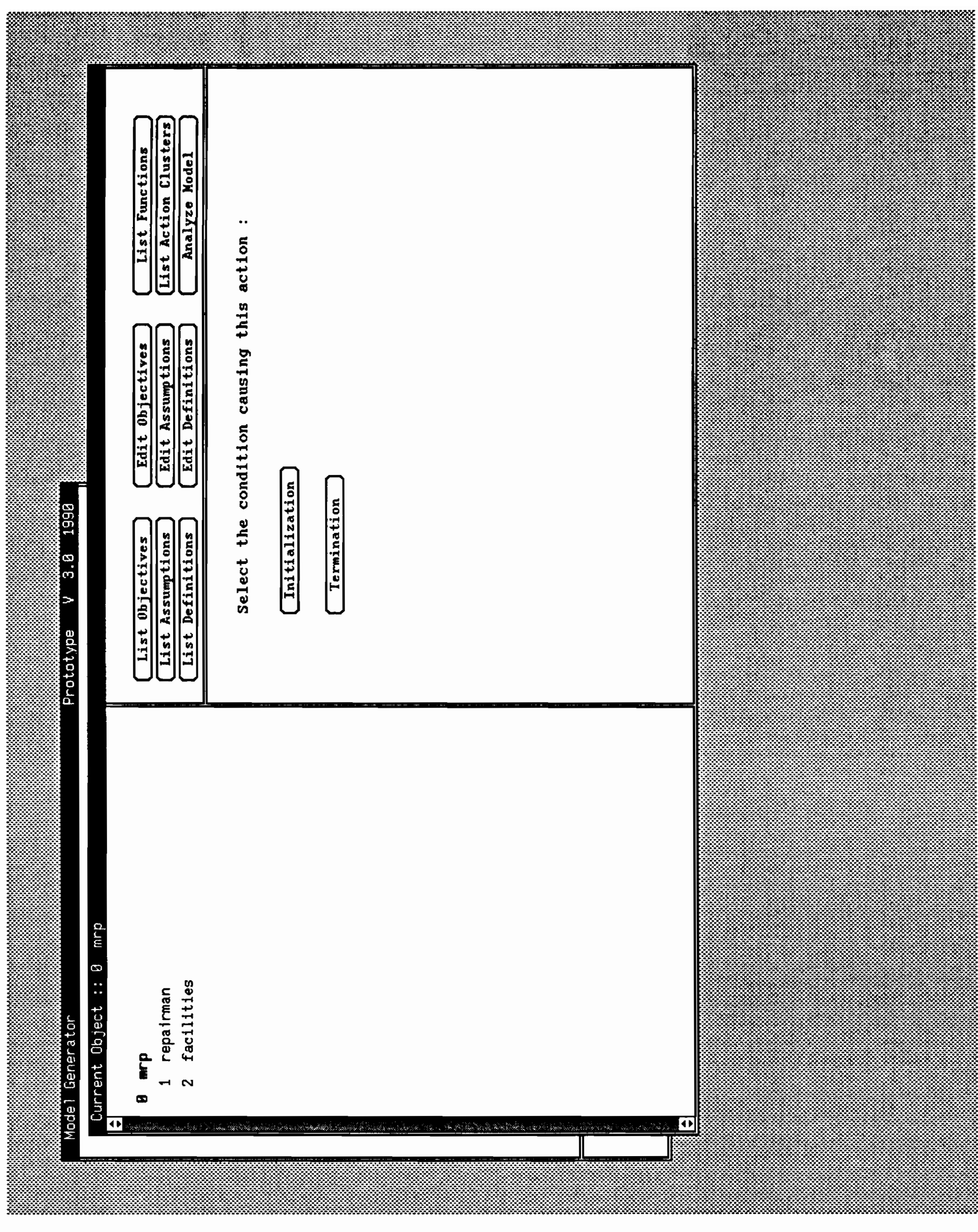

Figure 97. MG : Supply Condition Menu for Permanent Indicative Attributes.

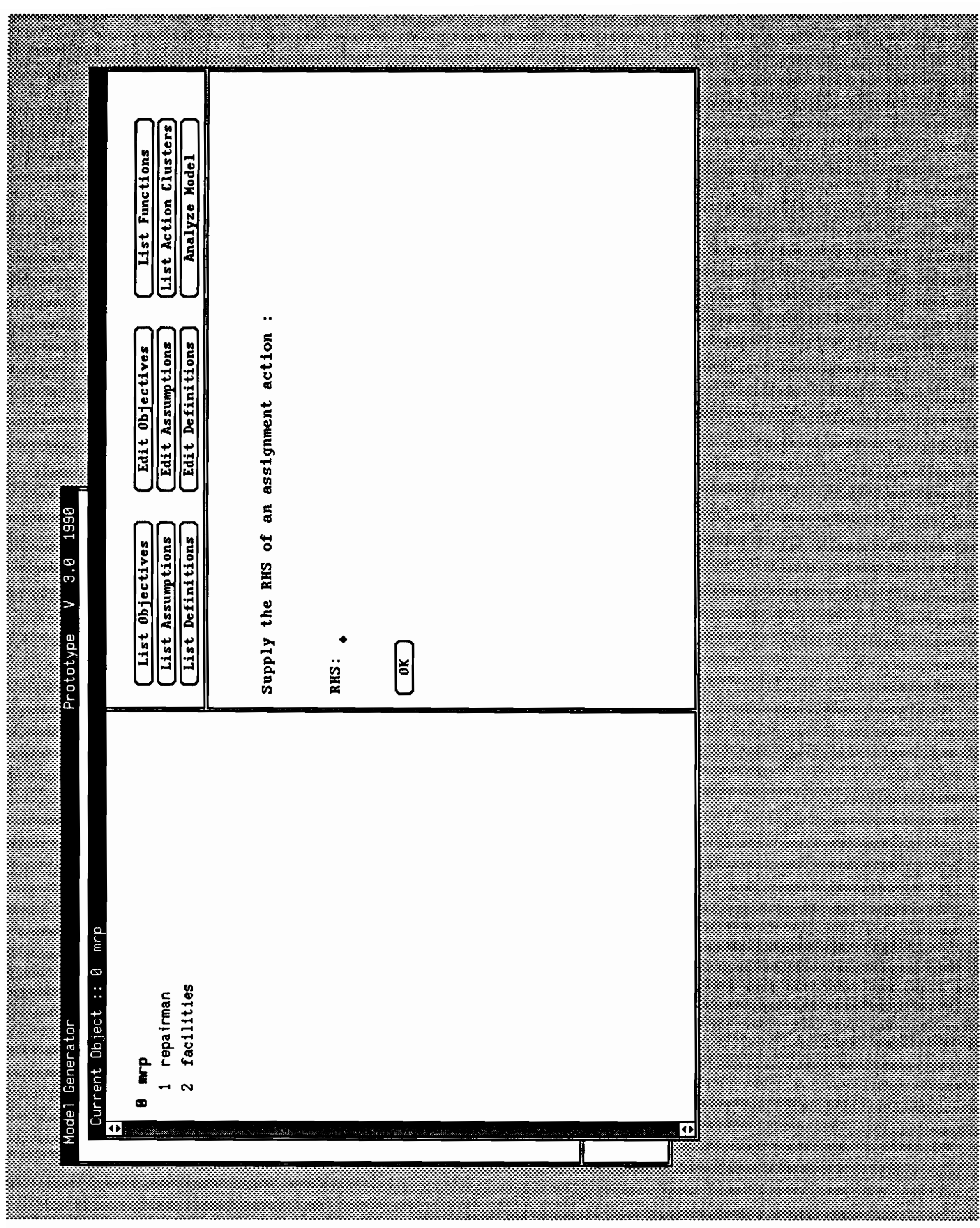

Figure 98. MG : Action Menu for Relational Attributes.

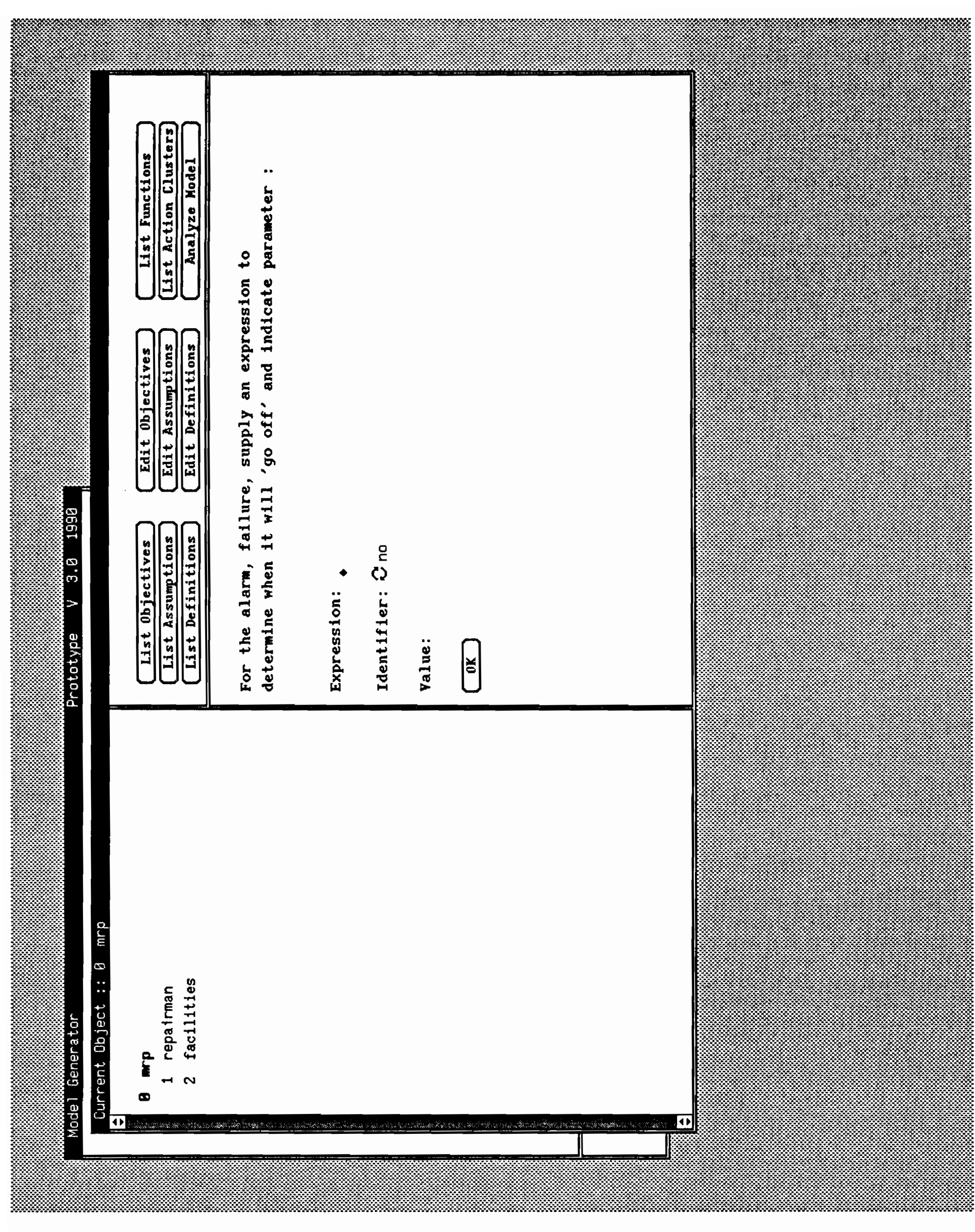

Figure 99. MG : Action Menu for Time-based Signal Attributes.

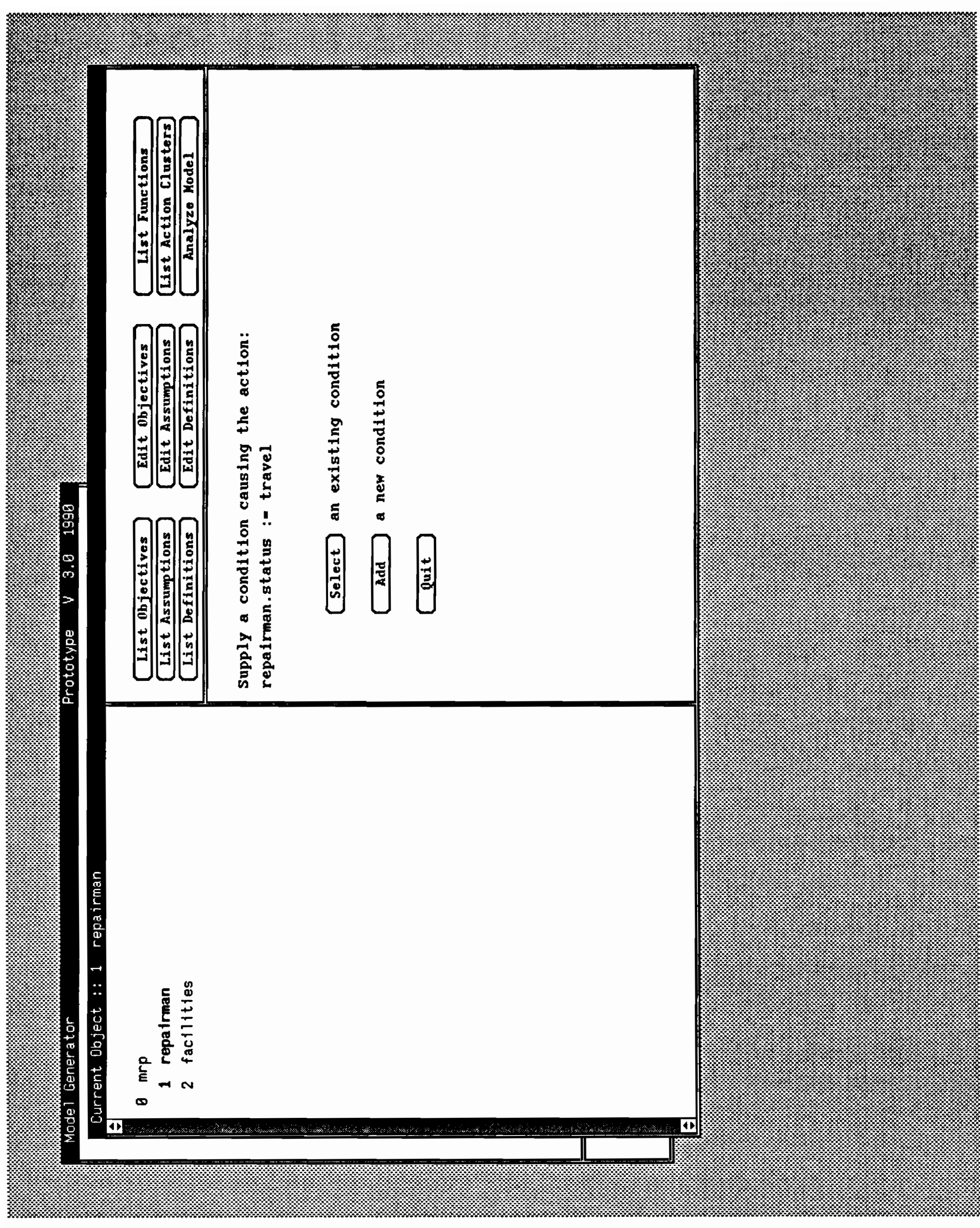

Figure 100. MG : Supply Condition Menu.

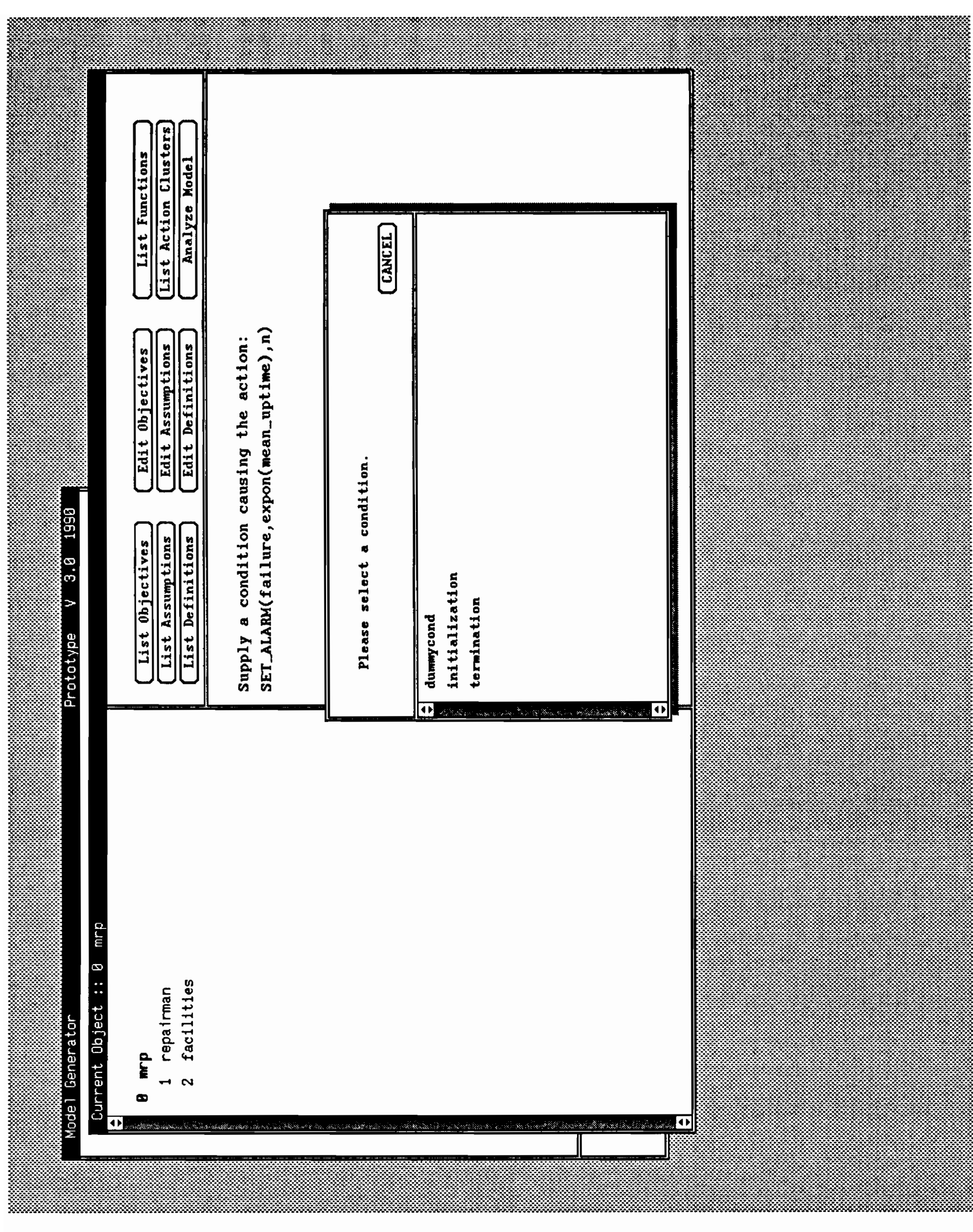

Figure 101. MG : Selecting an Existing Model Condition.

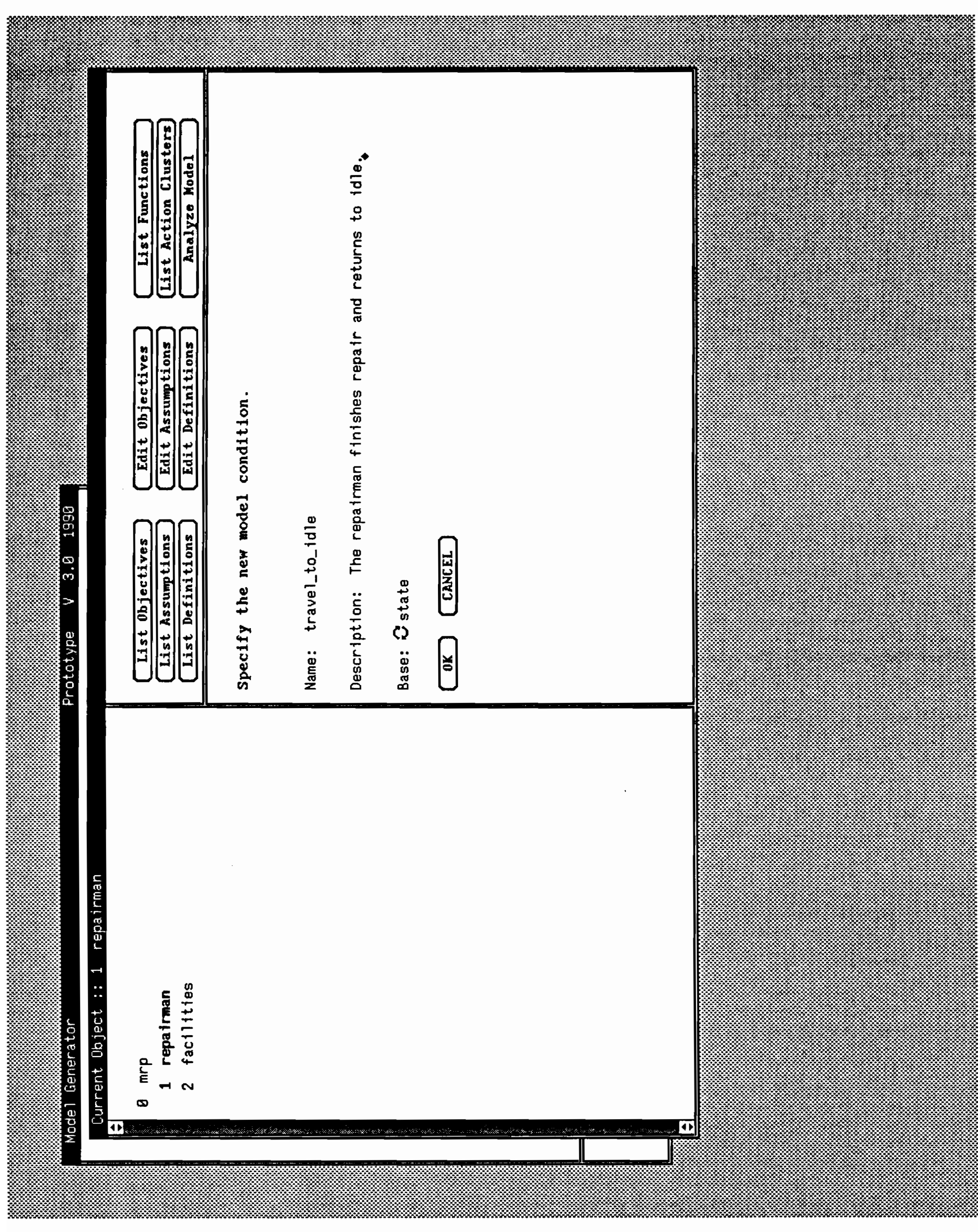

Figure 102. MG : Specifying a New Model Condition.

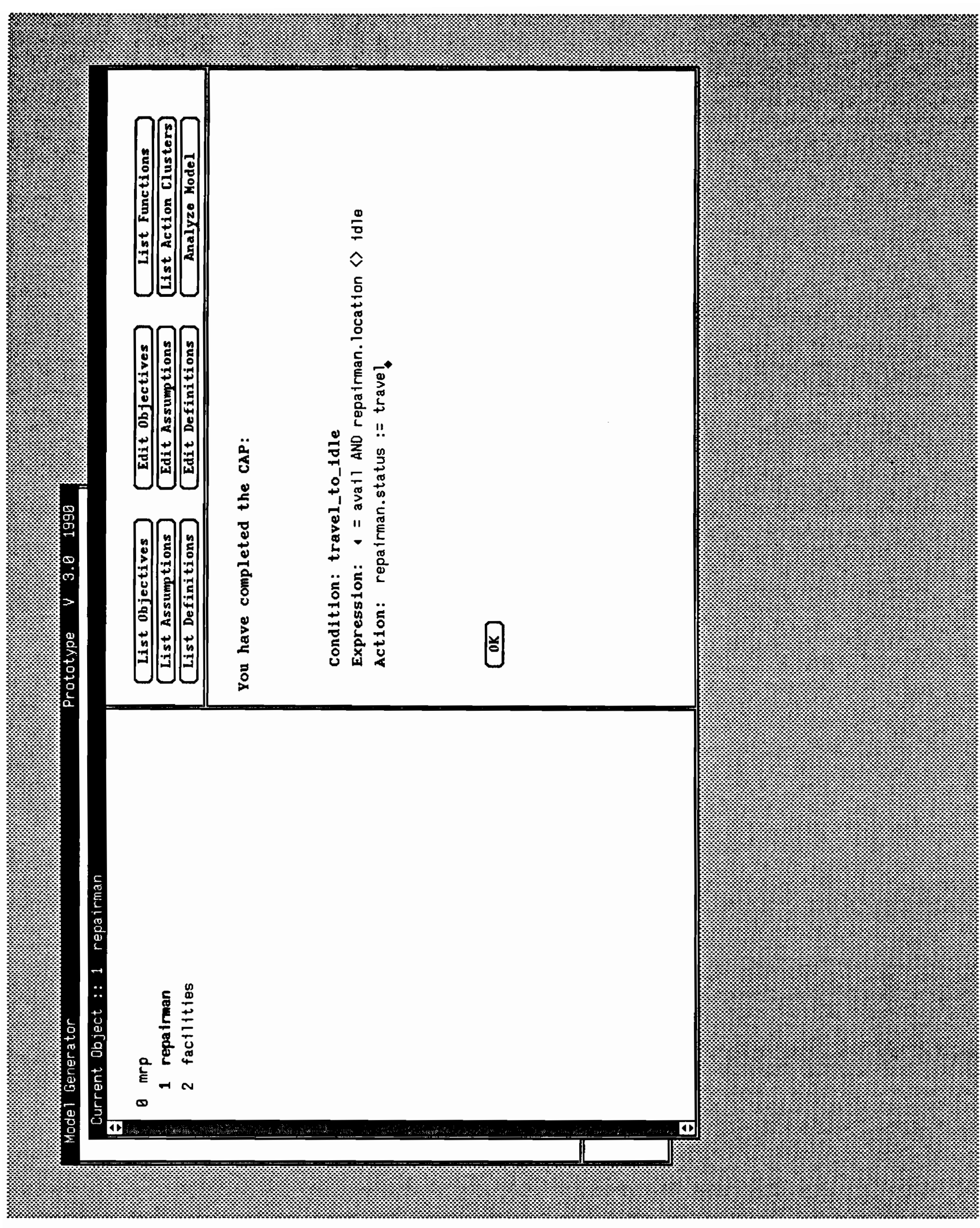

Figure 103. MG : The Completed CAP Display.

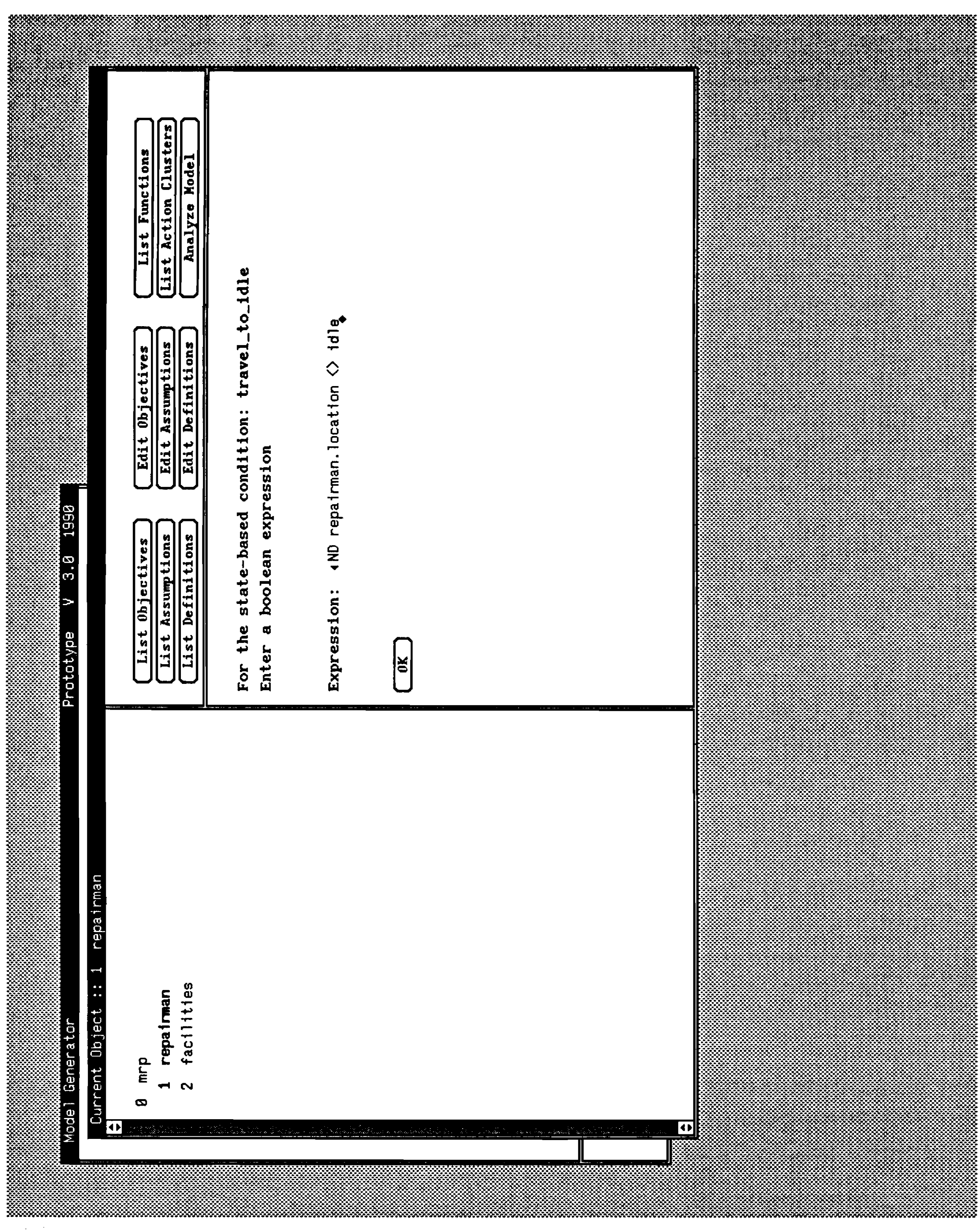

Figure 104. MG : Entering a Boolean Expression for a State-based Condition.

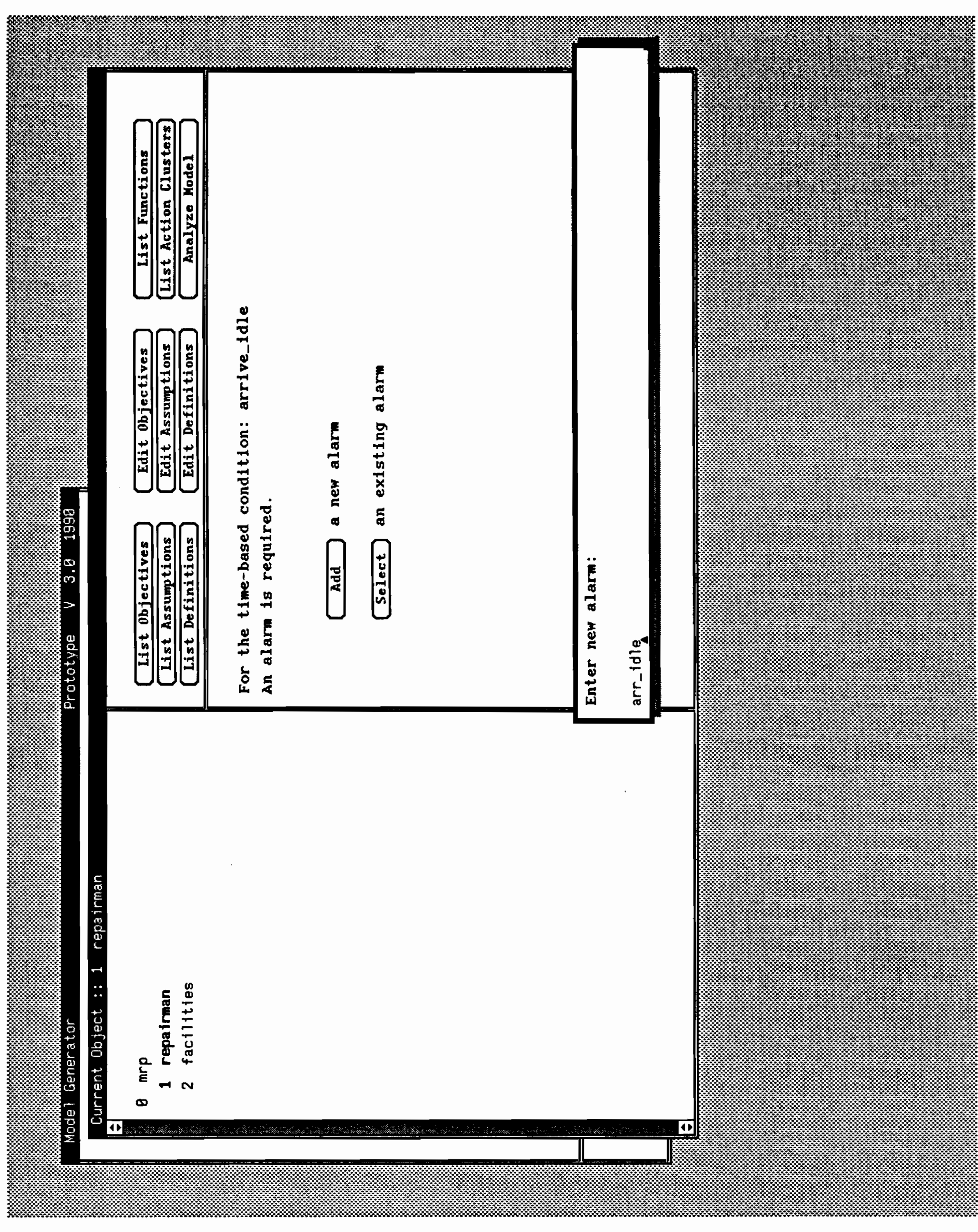

Figure 105. MG : Specifying an Alarm for a Time-based Condition.

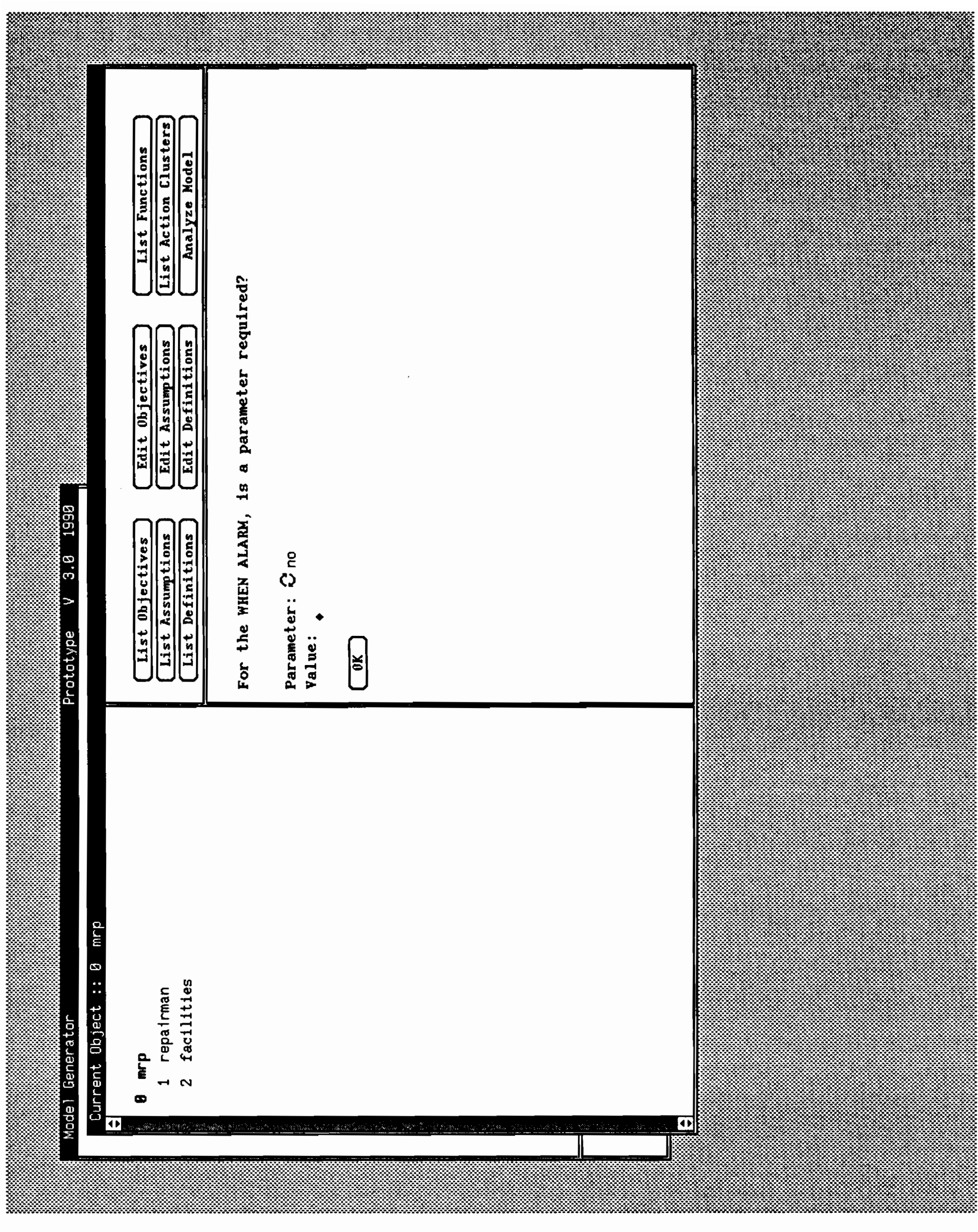

Figure 106. MG : Supply Parameter Menu for a Time-based or Mixed Condition.

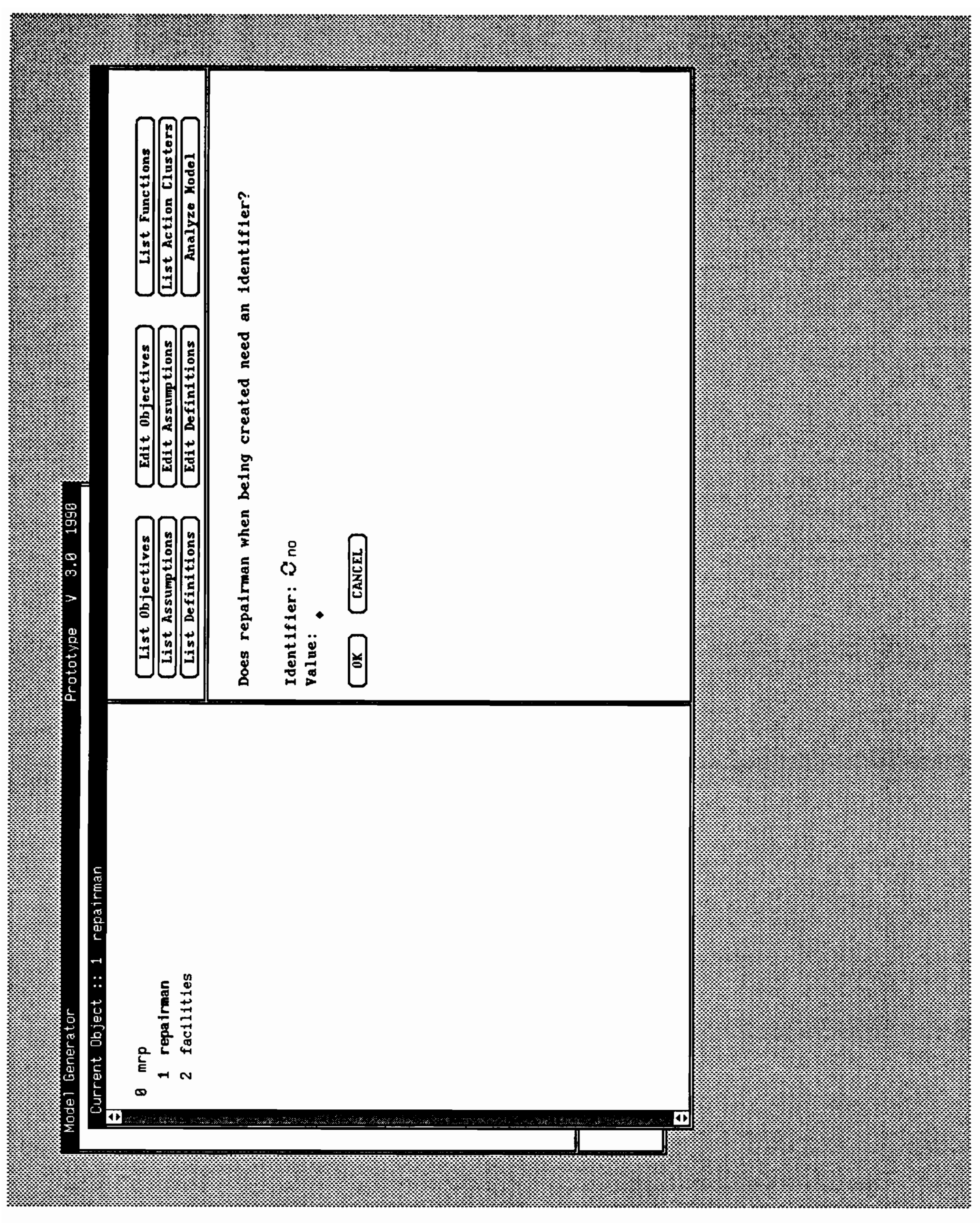

Figure 107. MG : Supply Identifier Menu for Object Creation and Destruction.

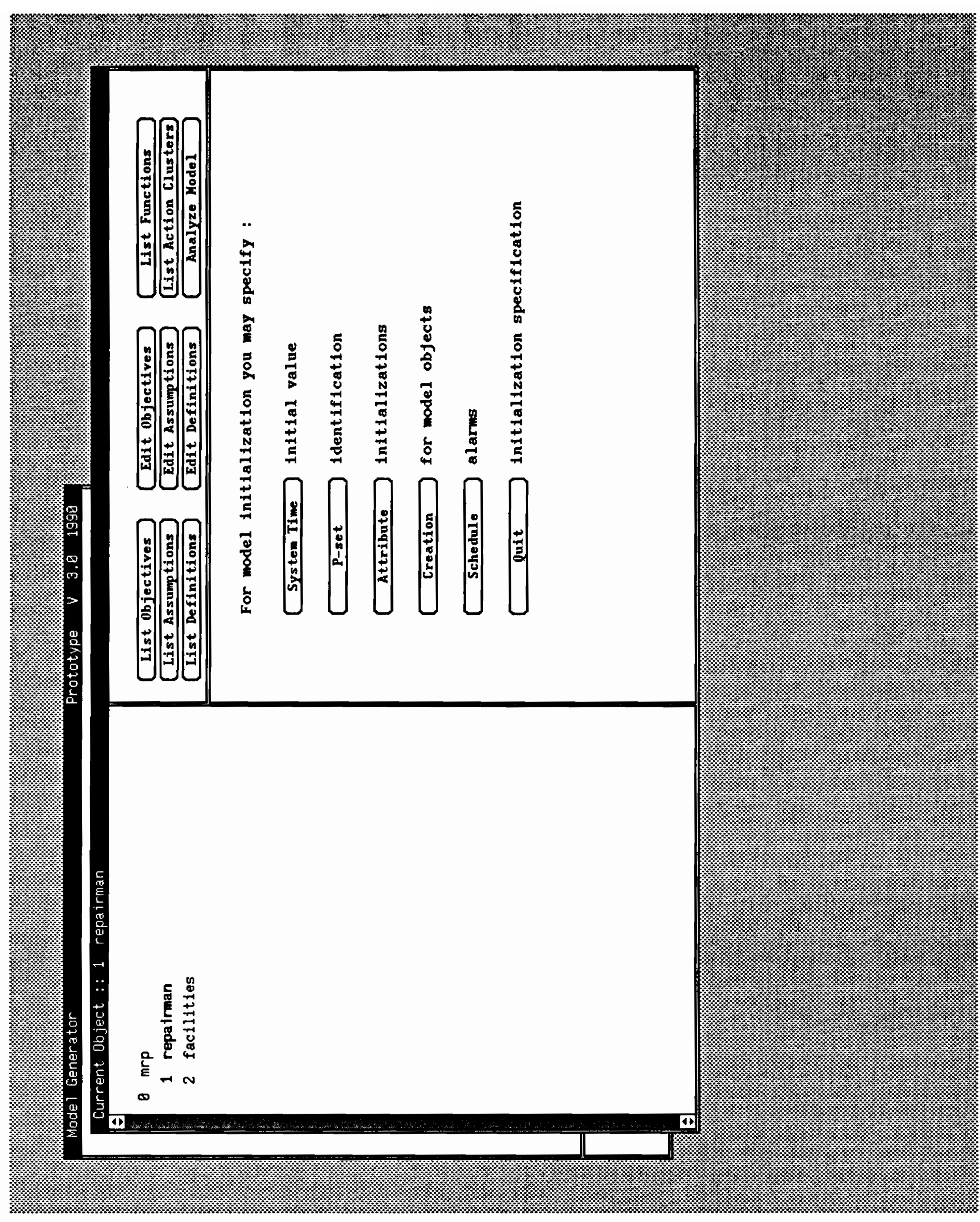

Figure 108. MG : Initialization Specification Menu.

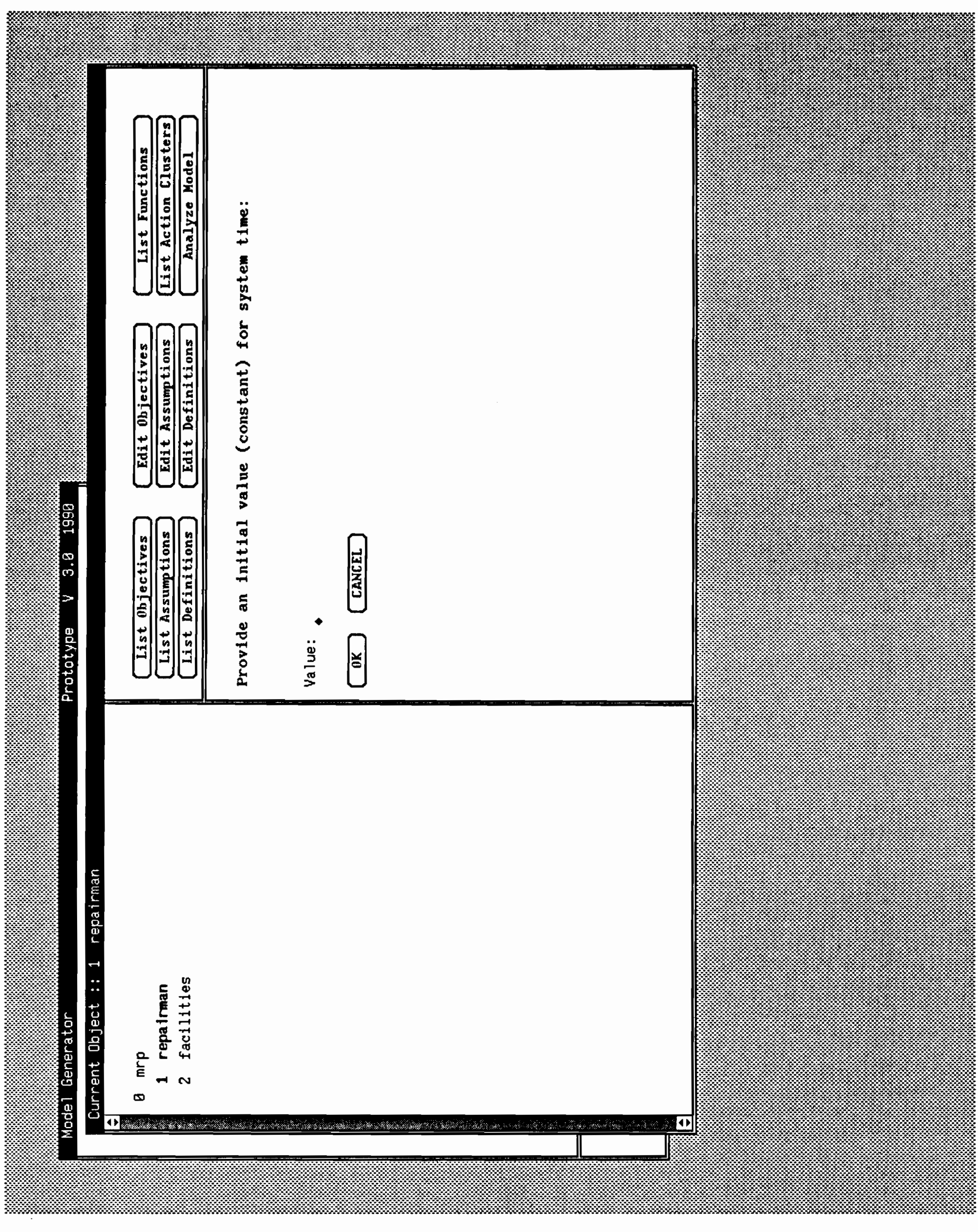

Figure 109. MG : Providing an Initial Value for System Time.

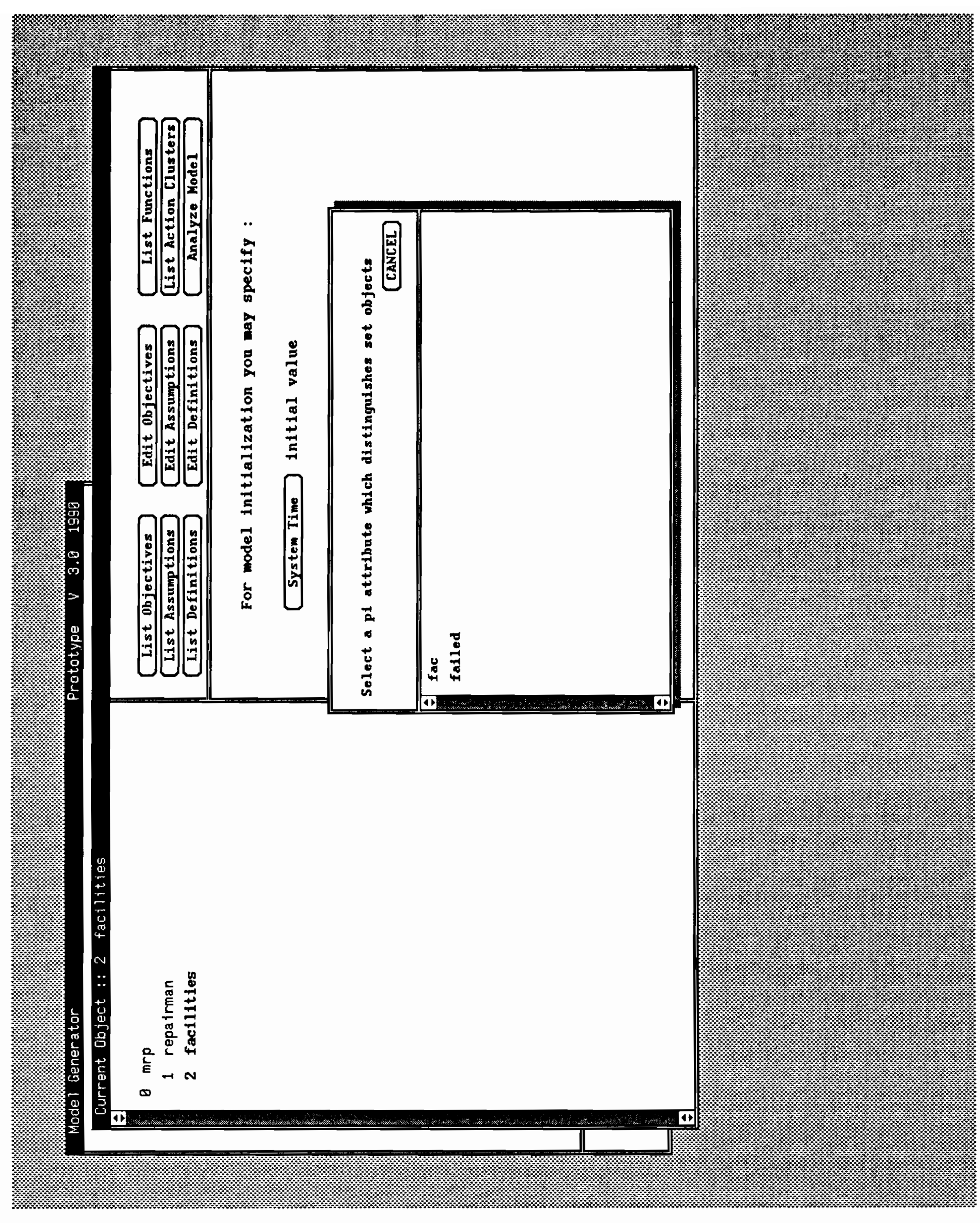

Figure 110. MG : Specifying Distinguishing Attribute for P-sets.

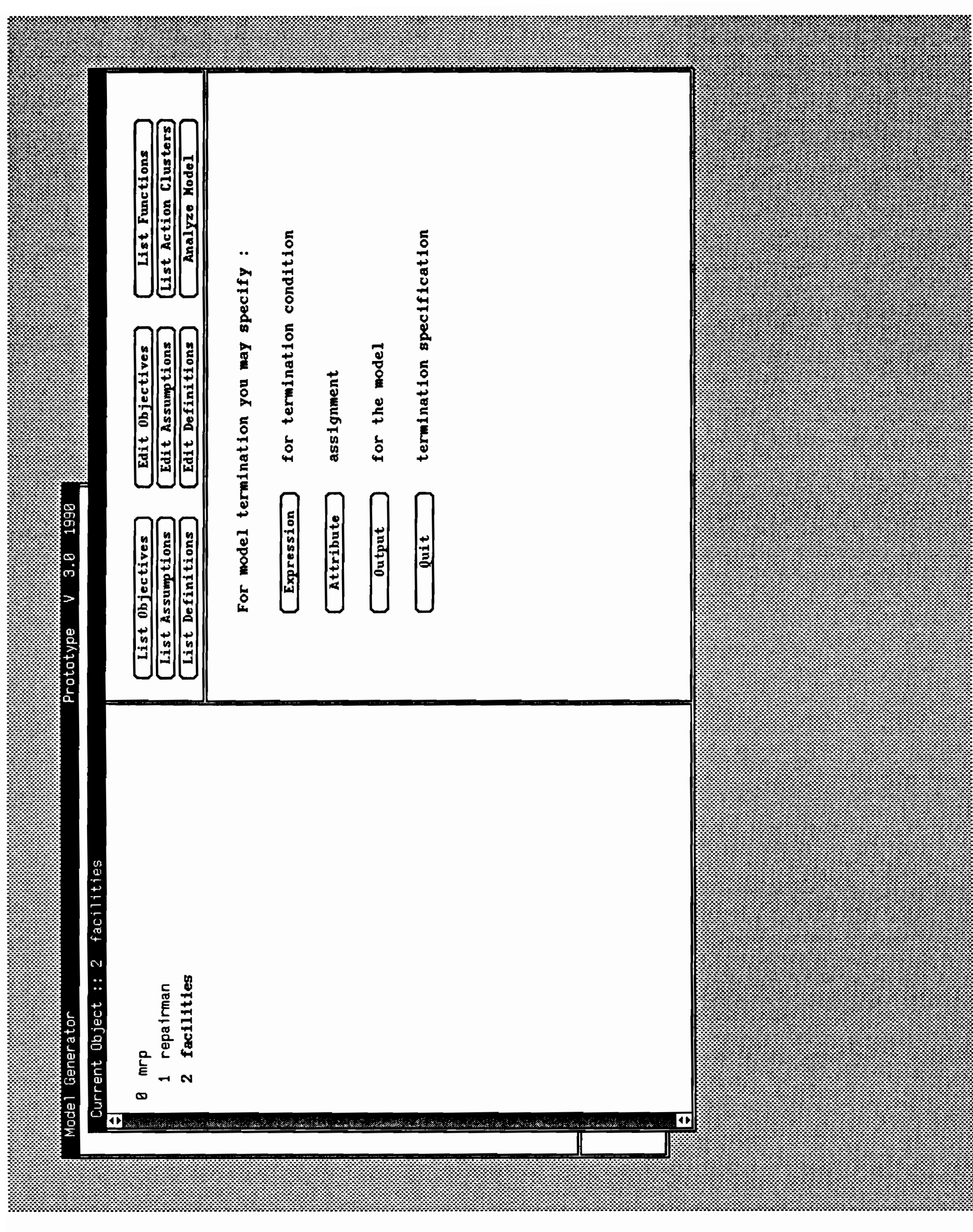

Figure 111. MG : Termination Specification Menu.

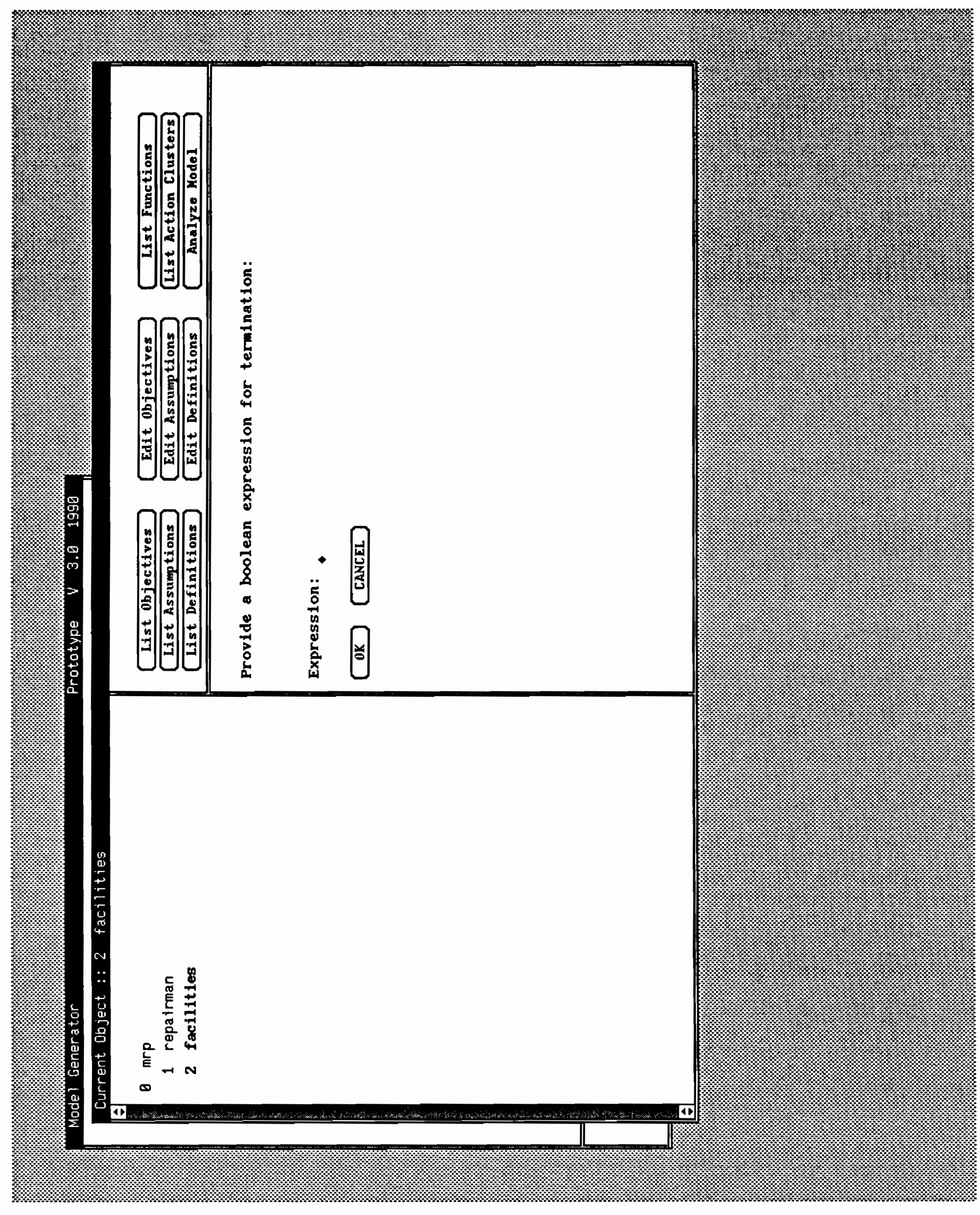

Figure 112. MG : Providing an Expression for Termination.

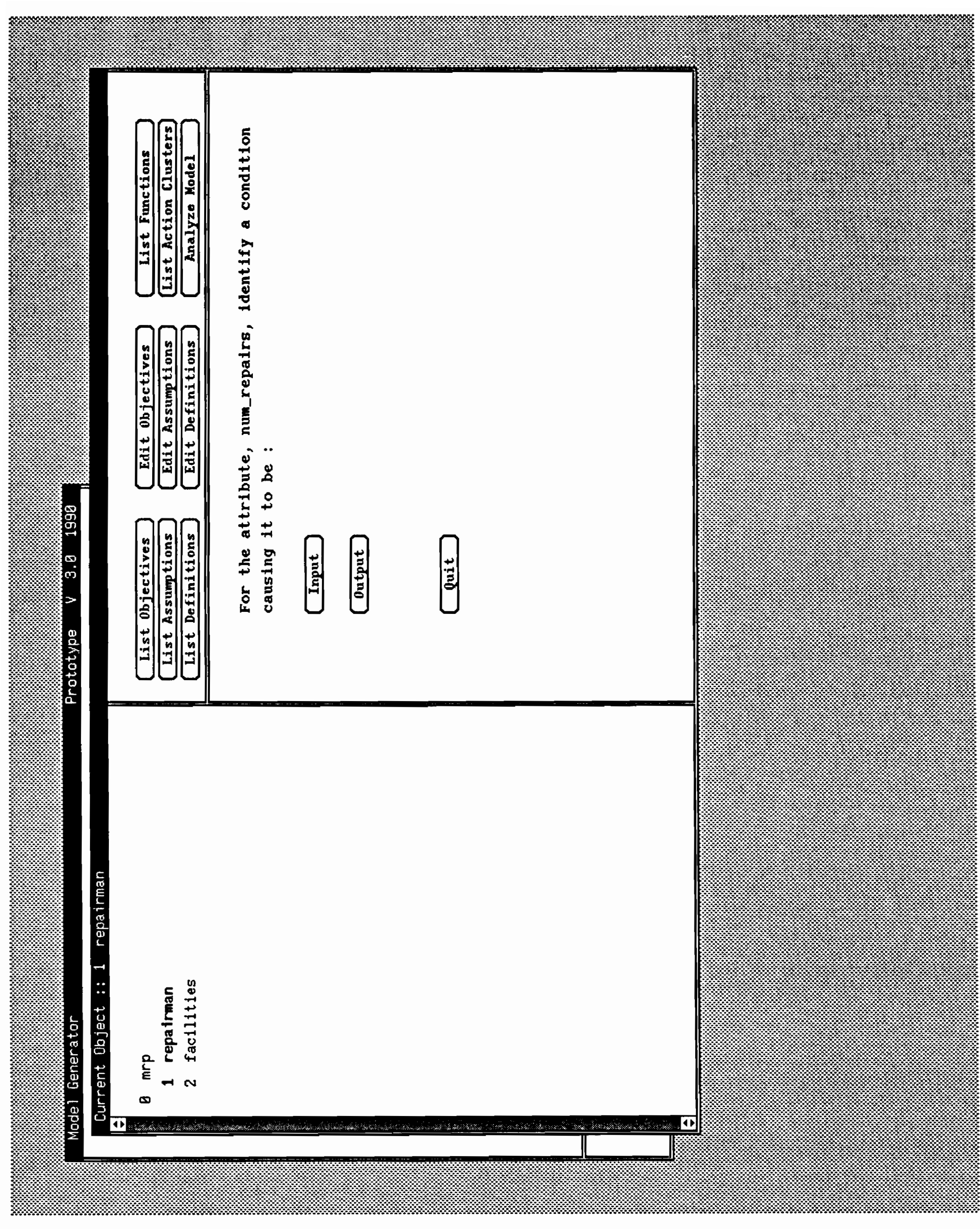

Figure 113. MG : Specify Input/Output Menu.

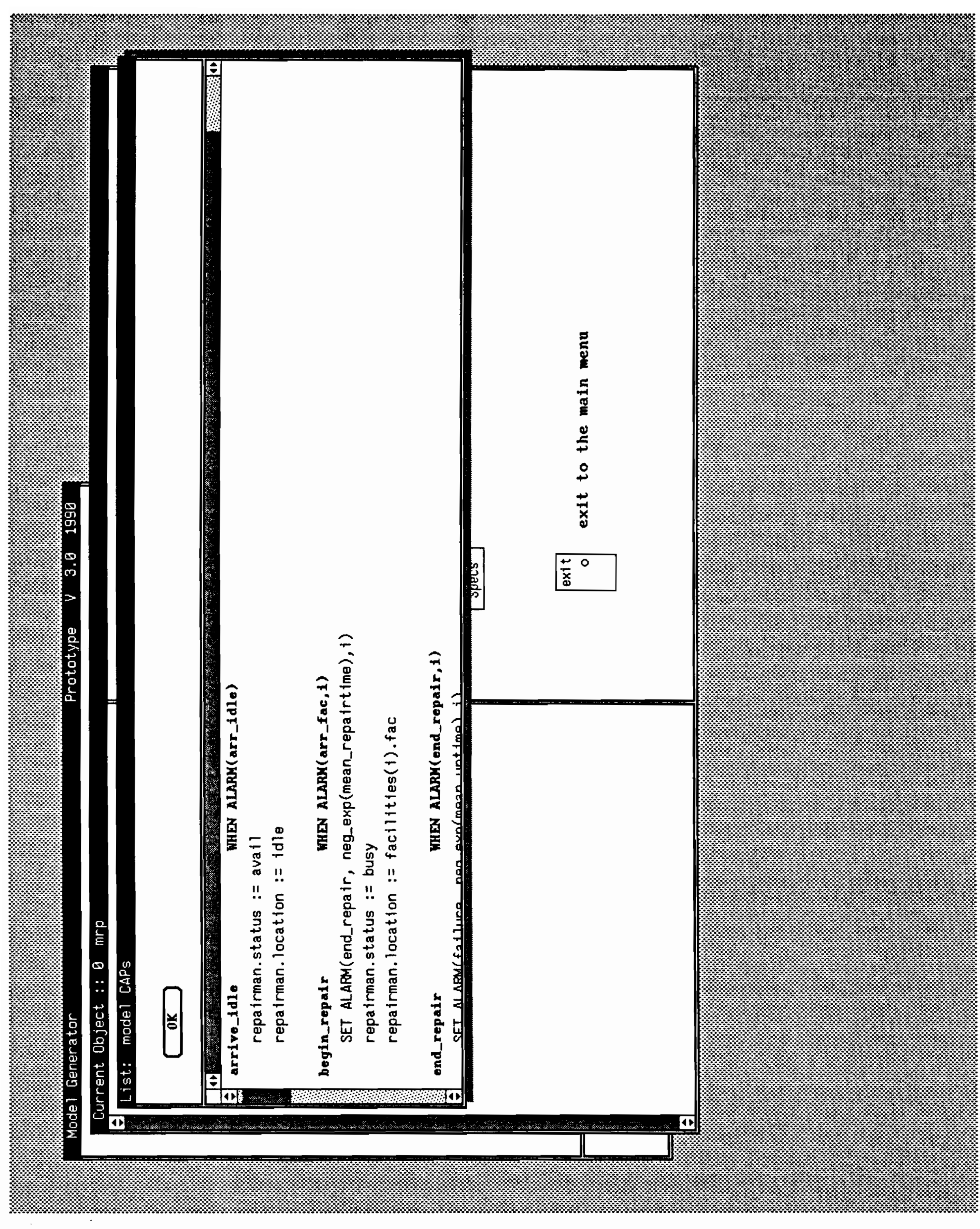

Figure 114. MG : Listing Action Clusters.

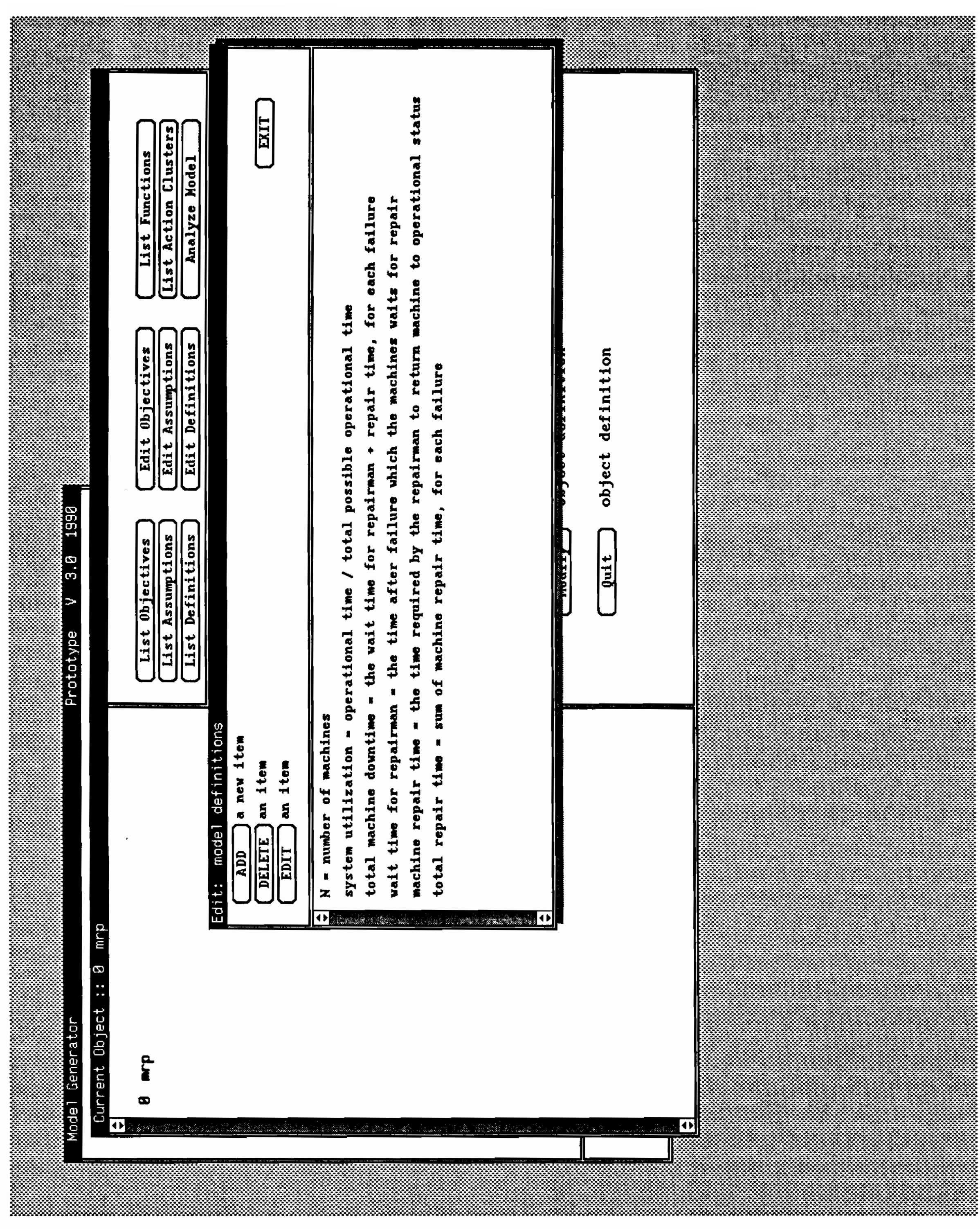

Figure 115. MG : Editing Model Documentation.

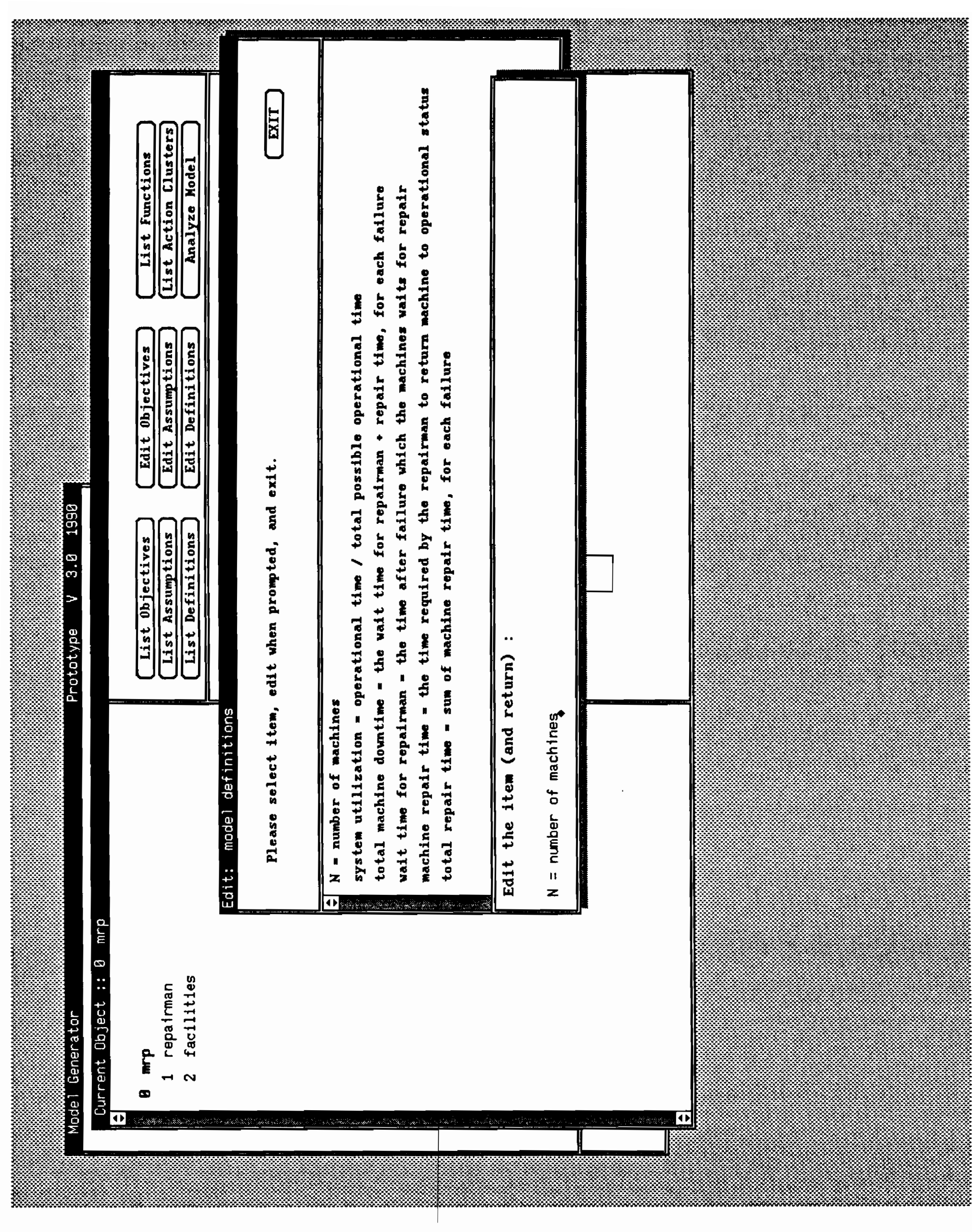

Figure 116. MG : Editing a Model List Item.

Vita

Name:	Ernest Henry Page, Jr.
Address:	12,400 I Foxridge Apts. Blacksburg, Virginia 24060
Phone:	(703) 552-9235
Birthdate:	7 February 1965
Marital Status:	Married, no children.

Education:

M.S. Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061
August 1990

B.S. Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061
February 1988

Ernest Henry Page, Jr. entered this world on a cold February morning in 1965 and found himself in the sleepy village of Alexandria, Virginia. He led a relatively quiet life there until 1968 when his father's work took the family to the even sleepier village of Pound, Virginia. In June of 1983 the author graduated from Pound High School and entered Virginia Tech to pursue a career in baseball. The author's exquisite mediocrity in the sport, however, forced him into an early retirement to concentrate on his studies. Further mediocrity, this time in Calculus, led the author to question this *latest* career move. Summer work roofing houses for Bill Baker, painting toilets for the Wise County School system, and shoveling asphalt for the Virginia Department of Highways and Transportation convinced the author that a college education was probably worth the effort, and further, that higher education might also be a good idea. Accordingly, after completing his B.S. in Computer Science during February of 1988 the author entered graduate school. Dr. R.E. Nance, a gentleman and scholar, offered the author research support with the SMDE project, which is where our tale finds him today. The author is currently residing in Blacksburg, Virginia pending the completion of his graduate studies and/or the rest of his life.

Ernest H. Page, Jr.